JavaScript Reference

Author: Michael de Raadt
University of Southern Queensland
Email: deraadt@usq.edu.au

Last Updated: Tuesday 31st May, 2005
Curriculum: A

1. First JavaScript Program

The process of writing a program in JavaScript is as follows.

1. Open a text editor (like Notepad)
2. Enter the JavaScript program within an HTML document
3. Save the document with a ".html" extension
4. Open the HTML document in a web browser.

When the browser opens the document it will run the JavaScript program or report errors in the code.

Writing programs is usually accomplished iteratively in small chunks. Start with a very basic program; save and open in a browser. Make a small change to the program, save then refresh the browser.

1.1. *Hello World!*

A traditional program to start programmers in a new language is one which outputs the message "Hello World!"

Exercise 1.1

Copy the following code (Code Example 1.1) into your text editor taking care not to introduce changes. The line numbers to the left of each line need not be entered; they are there so we can refer to a specific line in the code. Also, two symbols appear in the example which should be used as follows.

- Where the » symbol appears, press the TAB key; and
- Where the ¶ symbol appears, press the ENTER key.

In future examples, these symbols will not be shown explicitly, but will be used will be used when you write such code.

Save it as an HTML document with a filename like "hello.html"; open the document in your web browser.

```
01  <html>¶
02  »   <head>¶
03  »   »   <script type="text/javascript">¶
04  »   »   »   alert("Hello World!");¶
05  »   »   </script>¶
06  »   </head>¶
07  </html>¶
```

Code Example 1.1: Hello World!

1.2. *JavaScript and HTML*

HTML stands for HyperText Markup Language. Locating JavaScript code in an HTML document allows us to write and run simple programs easily.

An HTML document starts with an opening **`<html>`** tag and ends with a closing **`</html>`** tag. The document is divided into two parts.

The second part is the body, enclosed in **`<body>`**...**`</body>`** tags. The body contains text that will be shown in the browser window. Learning how to format and organised text in the body is interesting, but will not be covered here. In later examples we will use the body to add a label describing what JavaScript code we are testing.

The first part of an HTML document is the head, enclosed in **`<head>`**...**`</head>`** tags. The head contains extra information not shown in the browser window. It is here that we locate our JavaScript code. Within the head we add **`<script>`**...**`</script>`** tags to identify the start and end of our code. We also identify the scripting language used by adding the **`type="text/javascript"`** attribute to the starting **`<script>`** tag.

```
<html>
     <head>
          <script type="text/javascript">

                    JavaScript Code Here

          </script>
     </head>
     <body>

               HTML Here

     </body>
</html>
```

Code Example 1.2: JavaScript in an HTML document

Exercise 1.2

As we progress through this study, we will use the above format repeatedly. We will enter JavaScript code in the block identified by the label "JavaScript Code Here" and possibly add a simple label in the block identified by "HTML Here".

Using the file you created in Exercise 1.1 (called "hello.html") remove the line containing **`alert(...);`** and replace it with a blank line. This will be the section label "JavaScript Code Here" in Code Example 1.2.

Below the **`</head>`** tag, add a new line, press TAB and type **`<body>`**. Add a blank line and on another new line press TAB and then add a closing **`</body>`** tag. The blank line will be the section labelled "HTML Here" above. We will use this section to write a simple description of future programs.

Choose "Save As..." from the File menu and name the file "template.html". When creating a new JavaScript program, open the template and save it under a new name, then start adding code.

1.3. Statements

A JavaScript program is made up of *statements*. A statement usually starts at the beginning of a line and ends with a semi-colon (`;`). In Code Example 1.1 there is a single statement at line 04. Most programs have several statements.

2. Calling Functions

Built into JavaScript are a number of functions which achieve common, basic tasks like:

- Gathering input data from a user
- Displaying output to a user
- Discovering information about data
- Converting data from one form to another

To use a function, it is not necessary to know how it is constructed or how it achieves its task, you just need to know how to *call* the function. To call a function you need to know:

1. Its name (which very briefly describes the function's purpose)
2. What arguments (inputs to the function) are needed
3. What you might get back from the function

2.1. `alert()`

There are a number of functions a JavaScript program can use to get information to a user. One such function is **`alert()`**.This function takes a message and outputs it in a window that pops up within the user's browser. An example of such a window is show in Figure 2.1.

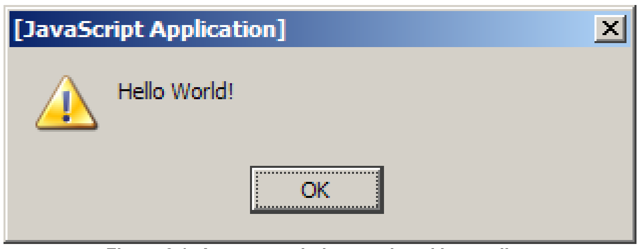


Figure 2.1: An output window produced by a call to `alert()`

In Code Example 1.1 (the first code example in this document) the **`alert()`** function is used on line 04 to produce the window above.

The **`alert()`** function has only one argument, the message to be output. As we will see later, it is possible to combine values together to form a single message output by this function.

Exercise 2.1

Open "template.html" and add a call to **`alert()`** in the JavaScript section. As an argument to the function, in between the parentheses **`()`** add your name surrounded by double quotes (`""`). Be sure the statement (the line) ends with a semi-colon (`;`).

In the HTML section (on the line after the opening **`<body>`** tag) Add text describing what the JavaScript program does. Text in this section is not part of the JavaScript program; only text between the opening **`<script>`** and closing **`</script>`** tags is regarded as JavaScript code. What text you add in the HTML section is up to you.

3. Values

In any programming language it is useful to distinguish different types of values so they can be treated differently in different circumstances.

3.1. Numbers

Numbers include integers (whole numbers like 1) and floating point numbers (numbers with a fraction after a decimal point like 1.23).

3.2. Strings

A string is a series of characters. A string can have many characters, a single character, or no characters at all (an empty string). To create a string, we use quotes to show the start and end of a string. Single or double quotes can be used as long as the same quotation mark is used at the start and end.

3.3. Booleans

There are only two Boolean values: **`true`** and **`false`**. These values do not need to be surrounded by quotes.

In Code Example 3.1 an example of each of the values above is output using **`alert()`**. On lines 04 and 05 strings are output using double and single quotes. On line 06 a number is output. On line 07 a Boolean value is output.

```
01 <html>
02      <head>
03          <script type="text/javascript">
04               alert("string content in double quotes");
05               alert('string content in single quotes');
06               alert(123);
07               alert(true);
08          </script>
09      </head>
10      <body>
11          Values example
12      </body>
13 </html>
```

Code Example 3.1: An example showing different values

Exercise 3.1

Using your template file, replicate Code Example 3.1. Does it output the values you expected it to?

Make the following changes.

1. On line 06 change the number from **`123`** to **`123.456`**. What happens?
2. On line 07 change the value of **`true`** to **`false`**. What happens?
3. Remove the quotes from around the string in the first call to **`alert()`** on line 04. What happens? Put the quotes back. What happens now?
4. Add another call to **`alert()`** with the argument **`abc`** (not in quotes). What happens?

4. Variables

4.1. What are Variables

A variable allows the storage of a value for later in the program. A variable can store any of the values shown in section 3. A variable is a piece of information named by an *identifier*. Where the identifier of a variable is located in a program, the value of the variable will be looked-up and used in its place.

4.2. Identifier Rules

There are some rules which constrain the identifiers you use.

1. Can contain:
   a. alphabetic characters **`A`** to **`Z`** and **`a`** to **`z`**
   b. numerals **`0`** to **`9`**
   c. underscores **`_`**
2. Cannot contain spaces, punctuation, quotes, or any characters not shown in 1 above.
3. Can start with:
   a. an alphabetic character **`A`** to **`Z`** and **`a`** to **`z`**
   b. an underscore **`_`**
4. Cannot start with a numeral or any character not shown in 3 above.

It should also be noted that identifiers are *case sensitive*, so a variable with an identifier **`userName`** will be a completely separate variable to one with an identifier **`UserName`**. Be careful; it is easy to accidentally misspell an identifier.

It is a good programming practice to use meaningful identifiers for variables. While it may be easier to name a variable **`x`** or **`myVar`**, such identifiers carry no description of the value they contain. It is better to identify a variable with a description of its contents. Multiple words can be used with second and subsequent words in the identifier staring with an uppercase letter. For example if one wished to store the name of the user, possible identifiers include **`nameOfUser`** or **`userName`**. Programmers tend to develop their own style for such aspects of programming and use the same style consistently.

Exercise 4.1

Are the following identifiers legal or not? If not, why not?

1. **`example-number`**
2. **`exampleNumber1`**
3. **`1exampleNumber`**
4. **`example_number`**
5. **`example number`**

4.3. Declaring Variables with `var`

If you wish to declare a variable, the best place to do this is at the start of your program. If you do this, the declaration will easy to find later.

The following form can be used to declare variables.

`var variableIdentifer;`

or

`var variableIdentifier = value;`

In the examples above **`variableIdentifier`** would be replaced by the identifier of the variable and **`value`** would be replaced with an initial value. Initialising variables will be discussed further in section 5. Examples of variable declarations are shown in Code Example 4.1.

```
01 <html>
02     <head>
03         <script type="text/javascript">
04             var exampleString = "string content in quotes";
05             var exampleNumber = 123;
06             var exampleBoolean = true;
07             var exampleVariable;
08
09             alert(exampleString);
10             alert(exampleVariable);
11         </script>
12     </head>
13     <body>
14         Variables example
15     </body>
16 </html>
```

Code Example 4.1: Declaring variables

4.4. Undefined

If a variable is declared and not given an initial value, it will not be given a default value. If a variable is given no value and an attempt is made to get the value out of the variable, the value **`undefined`** will be given. In Code Example 4.1 a variable is declared without an initialisation at line 07. In line 10 the value of the variable is accessed to be output. As the variable has not yet been assigned a value, the value **`undefined`** will be output.

Exercise 4.2

Before you start writing any code, look at Code Example 4.1. On a piece of paper, write what you think the program will output.

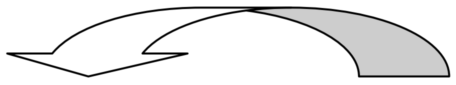
Using your template to replicate Code Example 4.1. Does it output the values as you expected it to?

Make the following changes.

1. Add another call to **`alert()`** to output **`exampleNumber`**.
2. Add another call to **`alert()`** to output **`exampleBoolean`**.
3. Create a new variable which will contain your name. Use an appropriately descriptive identifier which follows the rules shown in section 4.2. Assign the new variable a string (use quotes or double quotes) containing your name. Add another call to **`alert()`** to output the value of the variable.

5. Assigning Values

It is possible to give a variable a value when it is declared, and also to change its value later in the program. The form of an assignment is as follows.



`variableIdentifier = value;`

The value on the right is determined first. This could be from a number of sources. This value is then assigned to the variable identified on the left.

5.1. Dynamic Typing

Not only can the value of a variable change during the course of a program, but also the type of value may change. So a variable initialised with a string can later be assigned a number or a Boolean value. An example of this is show in Code Example 5.1.

```
<html>
     <head>
          <script type="text/javascript">
               var exampleVariable;

               exampleVariable = "string content in quotes";
               alert(exampleVariable);
               exampleVariable = 123;
               alert(exampleVariable);
               exampleVariable = true;
               alert(exampleVariable);
          </script>
     </head>
     <body>
          Dynamic typing example
     </body>
</html>
```

Code Example 5.1: The value and type of a variable can change

5.2. `typeof`

It is possible to determine if a variable currently contains a number, a string, a Boolean value, or no value at all (an **`undefined`** value). To do this, put the word **`typeof`** before the variable name (separated by a space). Code Example 5.2 is the same as the previous example, except instead of outputting the new values, the type of the variable is output at each stage.

```
<html>
     <head>
          <script type="text/javascript">
               var exampleVariable;

               alert(typeof exampleVariable);
               exampleVariable = "string content in quotes";
               alert(typeof exampleVariable);
               exampleVariable = 123;
               alert(typeof exampleVariable);
               exampleVariable = true;
               alert(typeof exampleVariable);
          </script>
     </head>
     <body>
          typeof() example
     </body>
</html>
```

Code Example 5.2: Discovering the type of a variable

Exercise 5.1

On a piece of paper write down the *identifier*, *value* and *type* of the following variables.

1. `var exampleInteger = 5;`
2. `var myName = "Michael";`
3. `var myLetter = 'M';`
4. `var emptyString = ""`
5. `var exampleTruthValue = true;`
6. `var exampleVariable;`

Exercise 5.2

On a piece of paper create variable declarations based on the following descriptions.

1. A number with identifier **`maxFound`** and value 0.
2. A string called **`name`** with your name as the value.
3. A Boolean variable called **`found`** with initial value **`false`**.

Exercise 5.3

Look at Code Example 5.2. On a piece of paper, write what you think the program will output.

Using your template to replicate Code Example 5.2. Does it output the values as you expected it to?

Make the following changes.

1. Change the double quotes on line 07 to single quotes. What happens?
2. Remove the contents of the string leaving only the quotes. What happens?
3. Change the number on line 09 to **`123.456`**. What happens?
4. Change the Boolean value on line 11 from **`true`** to **`false`**. What happens?

Not in implicit curriculum

5.3. Initialising Variables

When a variable is created its value is **`undefined`** until it is assigned a value. Using a variable that contains an undefined value can cause errors. Also, using value of one type (like a string) where another is expected (like a number) can have unexpected effects. It is therefore good practice to always initialise variables when they are created.

6. Operations

Operations are used to perform calculations and to combine values. There are four types of operators: arithmetic, relational, logical and string operators.

6.1. Arithmetic Operators

The form of operations you are probably most familiar with are arithmetic operations; operations on numbers. The following table describes the arithmetic operators available in JavaScript.

Operator	Name	Purpose
+	Plus	To add two numbers
-	Minus	To subtract one number from another
*	Multiply	To multiply two numbers
/	Divide	To divide two numbers
%	Mod	To find the remainder after integer division

Table 6.1: Arithmetic Operators

Each of the operators above can be used with two values (operands), one on each side. We call these *binary* operators. The Minus operator can also be used to negate the sign of a single variable from positive to negative and vice-versa. In this case we refer to the Minus operator as a *unary* operator.

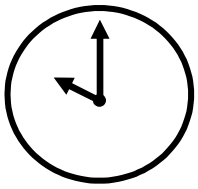
```
<html>
    <head>
        <script type="text/javascript">
            var number = 5;

            alert(1.2 + 3.4);
            alert(1.2 - 3.4);
            alert(1.2 * 3.4);
            alert(1.2 / 3.4);
            alert(12 % 5);
            alert(-number);
        </script>
    </head>
    <body>
        Operations example
    </body>
</html>
```

Code Example 6.1: Examples of arithmetic operators

The Mod (or Modulo) operator provides the remainder after a division. For example, say we had 12 apples and we wanted to divide this into groups of 5; how many would we have left-over? The 12 apples can give two full groups of 5 with 2 apples left-over.

Using the Mod operator we are able to bring large numbers to a position in a cycle. The Mod operator is sometimes called the clock operator. Consider a clock which shows the time at 10 o'clock. If asked what time will it be in 80 minutes, we do not say 10:80, we say it will be 11:20. We can use Mod to perform such a calculation as follows.



```
endMinute = (startMinute + minutesSpent)%60;
```

The Mod operator only works with whole numbers which we refer to as *integers*.

Exercise 6.1

On paper, write down what the following JavaScript statements will output.

1. `alert(1 + 2.5);`
2. `alert(1 - 2.5);`
3. `alert(2 * 3);`
4. `alert(1 / 2);`
5. `alert(5 % 3);`
6. `alert(9 % 3);`

Not in implicit curriculum

6.2. Division by Zero – `infinity`

When a number is divided by zero, the mathematical result is irrational. In JavaScript when a number is divided by zero, the special value **`infinity`** is given as the result. Care must be taken to avoid using **`infinity`** later in another operation as this may crash your program.

```
<html>
     <head>
          <script type="text/javascript">
               alert(123 / 0);
          </script>
     </head>
     <body>
          Division by zero example
     </body>
</html>
```

Code Example 6.2: Dividing by zero results in infinity

6.3. Postfix Operators

A common operation is increasing a variable's value by one (*increment*) or reducing its value by one (*decrement*). One way to achieve an increment as follows.

```
numberVariable = numberVariable +1;
```

A simpler short form is provided using the *unary* **`++`** operator.

```
numberVariable++;
```

A similar operator (**`--`**) is provided for decrementing. Both operators are demonstrated in Code Example 6.3.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 0;

               alert(number);
               number++;
               alert(number);
               number--;
               alert(number);
          </script>
     </head>
     <body>
          Postfix operations example
     </body>
</html>
```

Code Example 6.3: Increment and decrement operators

Exercise 6.2

On paper, write down what the Code Example 6.3 will output.

6.4. *Relational Operators (incl. Equality)*

A relational operator takes two values (usually numbers) and compares them. The result of such an operation will be a Boolean value of **`true`** or **`false`**.

Operator	Name	How it works
>	Greater than	**`true`** if left value is greater than right
>=	Greater or equal	**`true`** if left value is equal or greater than right
<	Less than	**`true`** if left value is less than right
<=	Less or equal	**`true`** if left value is equal or less than right
==	Equal	**`true`** if left and right values are equal
!=	Not equal	**`true`** if left and right values are not equal

Table 6.2: Relational Operators

Examples of relational operators are shown in Code Example 6.4.

```
<html>
     <head>
          <script type="text/javascript">
               var x = 1;
               var y = 2;

               alert(x > y);
               alert(x >= y);
               alert(x < y);
               alert(x <= y);
               alert(x == y);
               alert(x != y);
          </script>
     </head>
     <body>
          Relational operations example
     </body>
</html>
```

Code Example 6.4: Relational operators

Exercise 6.3

On paper, write down what Code Example 6.4 will output.

6.5. *Logical Operators*

Logical operators combine two Boolean values. The resulting value will be **`true`** or **`false`**.

Operator	Name	How it works
&&	And	**`true`** if both values are **`true`**
||	Or	**`true`** if one or both values are **`true`**
!	Negate	**`true`** becomes **`false`**, **`false`** becomes **`true`**

Table 6.3:Logical Operators

Examples of relational operations are show in Code Example 6.5.

```
<html>
     <head>
          <script type="text/javascript">
               var x = 1;
               var y = 2;
               var testValue = false;

               alert(x==1 && y==2);
               alert(x==1 && y==1);
               alert(x==1 || y==1);
               alert(x==0 || y==0);

               testValue = x>0;
               alert(testValue);
               alert(!testValue);
          </script>
     </head>
     <body>
          Logical operations example
     </body>
</html>
```

Code Example 6.5: Logical operators

Exercise 6.4

On paper, write down what Code Example 6.5 will output.

6.6. String Operators

The **+** operator can be used to join two strings. It can also be used to append other values (numbers or Booleans) to the end of a string.

```
<html>
     <head>
          <script type="text/javascript">
               var message = "Hello";
               var number = 5;

               message = message + " World!";
               alert(message);
               alert("Mambo No. " + number);
          </script>
     </head>
     <body>
          String operations example
     </body>
</html>
```

Code Example 6.6: String operations

Exercise 6.5

On paper, write the following program (you only need the part that goes between the **`<script>...</script>`** tags.)

1. Declare a variable called **`message`** and initialise it with the string **`"Hello"`**.
2. Add a space to the end of the string value using a **+** operation.
3. In a call to **`alert()`** output the value of message and append your name as a string in quotes.

Not in implicit curriculum

7. Abutment

Most computers can only achieve one action at a time. With modern operating systems, computers can run multiple programs at the same time, but actually these programs must take turns accessing the computer's processor to complete their next action. Within programs, only a single statement can be processed at a time. Statements are processed in order from top to bottom. It is therefore important to recognise that to achieve a certain goal or goals, the steps required to achieve this must be discovered and the order in which they are put into action must be understood.

Take, for example, the simple goal of adding two numbers for a user. We can plan the steps involved as follows.

1. Declare two variables
2. Input two numbers
3. Perform calculation
4. Output result

To complete the required goal, the steps above cannot be ordered in any other way. In a program each of the steps will be performed in order and never out of sequence. Placing these steps adjacent to each other, one after the other, is referred to as *abutment*.

If this goal were part of some larger goal, the simple plan shown above would need to be *abutted* with other plans.

Exercise 7.1

On paper, *order* the following steps to create the message "Hello XXX" for a user who's name will replace XXX.

a. Append user's name to message variable.
b. Declare a message variable initialised to "Hello "
c. Get the user's name and assign to **`userName`**.
d. Output message.
e. Declare a variable with identifier **`userName`**.

8. Debugging

An important skill in programming is to find problems on code that:

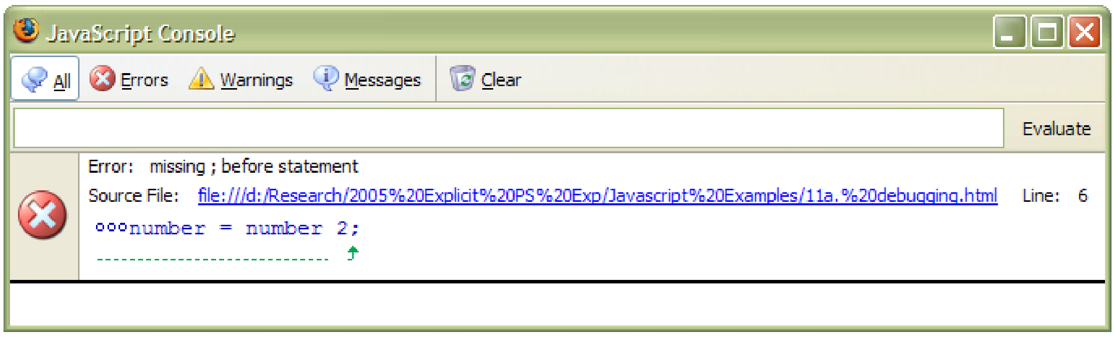
1. Stop the program from running at all, or
2. Don't stop the program running, but cause the program to perform incorrectly.

When writing code, you will be initially concerned with the first of these two.

```
01 <html>
02      <head>
03          <script type="text/javascript">
04              var number = 1;
05              number = number + 1;
06              number = number 2;
07              number = number * 3;
08          </script>
09      </head>
10      <body>
11          Debugging example
12      </body>
13 </html>
```

Code Example 8.1: A program containing a bug (line 06)

The code you write in JavaScript is interpreted by the Web Browser that is displaying the page. Different Web Browsers will deal with bugs in JavaScript code in different ways. The code in Code Example 8.1 contains an error on line 06; an operator is missing between the variable identifier **`number`** and the value **`2`**. After reaching this point in the program, the Web Browser would stop and the remaining program will not be executed. Using Mozilla Firefox (v1.0) the JavaScript Console reports errors. The JavaScript Console can be accessed from the Tools menu.



```
Error:   missing ; before statement
Source File:  file:///d:/Research/2005%20Explicit%20PS%20Exp/Javascript%20Examples/11a.%20debugging.html   Line:  6
ooonumber = number 2;
---------------------------- ↑
```

Figure 8.1: The JavaScript Console from Mozilla Firefox v1.0

Try to work on one error at a time. Error messages are the Web Browsers best guess at the program author's intention. Quite often they are incorrect and often confusing. What we can determine is:

- What line the error appeared on, and
- Roughly where in the line the error was located.

Knowing where the error has occurred is a good start. Return to the source code of the program and find the location. Sometimes the error is obvious and relying on what you have learned so far, it should be possible to correct the error. If the error does not jump out at you, and you find yourself staring indefinitely, ask for help.

When you have changed the source code, save the file, go to the JavaScript console and press the "Clear" button, then go to the Web Browser and click "Refresh". On returning to the JavaScript Console, hopefully the error will be gone.

```
01 <html>
02     <head>
03         <script type="text/javascript">
04             var exampleString = a string value;
05             var number 1;
06 
07             alert(exampleString " and some more");
08         </script>
09     </head>
10     <body>
11         Debugging example for exercise
12     </body>
13 </html>
```

Code Example 8.2: A program containing several bugs

Exercise 8.1

The code in Code Example 8.2 contains three errors. Before you enter the code into your computer, attempt (on paper) to identify the line numbers containing errors, give a description of the error and say how you would fix it.

Using your template, enter the code exactly as shown. Open the file in your Web browser. Open the JavaScript Console (Tools → JavaScript Console) and attempt to locate and fix the errors one at a time. If you get stuck, ask for help.

A strategy for discovering faults in a program that is running but produces incorrect results is referred to as "print-lining". As a program runs, the variables in the program change. If the end result is incorrect, the point at which the program deviated from your intended route needs to be discovered. At points in your program it is possible to add calls to the **`alert()`** function to output the value of a variable (or variables) at that point. Usually it is best to start near the beginning, moving the line containing the call to **`alert()`** to later points in the program until the place where things start to go wrong is identified.

Not in implicit curriculum

```
<html>
    <head>
        <script type="text/javascript">
            var exampleVariable;

            alert(3 + exampleVariable);
        </script>
    </head>
</html>
```

Code Example 8.3: Code contains an error, but where

Exercise 8.2

Code Example 8.3 contains an error. Use your template to create this example.

- What does this program produce as output?
- Use a call to **`alert()`** to output the value of **`exampleVariable`** before line 06.
- What is the error?
- What can be done to remedy the error?

9. Functions that Return Values

The **`alert()`** function produces output to the user, but does not gather or create any information that can be used in our program. Many functions in JavaScript perform some action, then return a value that can be used in your program.

9.1. `prompt()`

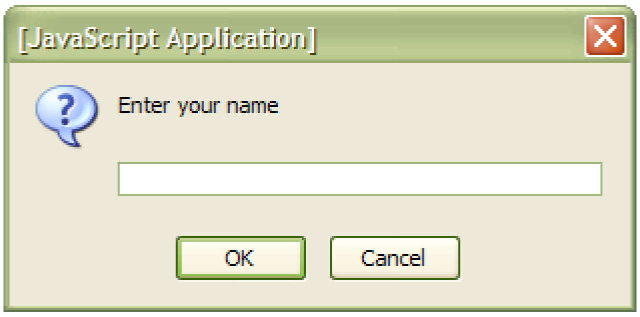
The function **`prompt()`** is an example of a function which returns a value. This function, as the name implies, prompts the user to enter some information. That information is captured and can then be used in the program. The function **`prompt()`** returns a string value. To use this string we can either:

- Store the value in a variable;
- Use the value in an operation; or
- Pass the value on to another function as an argument (input).

```
01 <html>
02      <head>
03           <script type="text/javascript">
04                var name = "";
05 
06                name = prompt("Enter your name");
07                alert("Hello " + name);
08           </script>
09      </head>
10      <body>
11           Input example
12      </body>
   </html>
```

Code Example 9.1: Example using the `prompt()` function

In Code Example 9.1 the **`prompt()`** function is used at line 05. The user is prompted to enter their name. A text box is given to do this as shown in Figure 2.1. When complete the user presses the ENTER key or clicks the OK button.


[JavaScript Application]

Enter your name

Figure 9.1: The effect of a call to `prompt()`

Still on line 05 of Code Example 9.1 the string returned from **`prompt()`** (the string entered by the user) is stored in the variable **`name`**. On the next line, the message "Hello" followed by the name the user entered is output.

Exercise 9.1

Using your template, create a program that will ask the user their age using the **`prompt()`** function. Store the age in a variable called **`age`**. Output the message "You are XXX years old" where XXX is the age entered by the user.

9.2. `parseInt()` and `parseFloat()`

The function **`prompt()`** returns a string. This is good where a string is needed, but a string cannot be used in arithmetic operations. Two functions are provided which can take a value (including a string) and turn it into a number. The function **`parseFloat()`** will return a number with a fraction expressed in decimal places. The function **`parseInt()`** will return a number without any decimal places. It should be noted that it does not round a number, it truncates it (just chops off the decimal places). So sending the value 1.9 to **`parseInt()`** would result in a value of 1.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 0;

               number = prompt("Enter a number");
               alert(typeof number + " " + number);
               number = parseInt(prompt("Enter a number"));
               alert(typeof number + " " + number);
               number = parseFloat(prompt("Enter a number"));
               alert(typeof number + " " + number);
          </script>
     </head>
     <body>
          parseInt() and parseFloat() example
     </body>
</html>
```

Code Example 9.2: Using `parseInt()` and `parseFloat()` to get an numeric input

Exercise 9.2

Answer to the following on paper first, then confirm your answer by creating and testing the program. Assuming a user entered 4.56 for each input, what would the program in Code Example 9.2 output?

Exercise 9.3

Using the code you wrote in Exercise 9.1 take the result returned by the **`prompt()`** function and pass it to **`parseInt()`** to convert the user's age from a string to a number in integer form and store this in **`age`**. Increment the value of **`age`**. Output the message to say "Soon you will be XXX years old" where XXX is the incremented age.

10. Selection

In the programs we have seen so far, there has been only one path of execution through the program. Sometimes we may wish to execute some statements only when certain conditions are met. Sometimes we may wish to have two possible sets of statements of which only one will be executed according to certain conditions. Choosing whether or not to execute a body of statements is referred to as *selection*. A number of structures are provided for us to achieve selection.

10.1. The `if` Statement

The **`if`** statement can be used to execute a *body* of statements when certain conditions are met. We use a *test* to determine if these conditions have been met. The test will result in a **`true`** or **`false`** value. Relational (**`>`**, **`<`**, **`==`**...) and logical (**`&&`**, **`||`**, **`!`**) operators are often used in such a test to obtain a Boolean value.

```
if( TEST ) {
        BODY
}
```

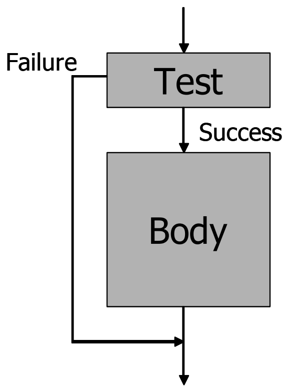


Figure 10.1: How `if` works

In the syntax description above we see an **`if`** statement starting with the word **`if`**. This is followed by the test which is always enclosed in parentheses **`()`**. The body contains statements that will be executed if the test results in a **`true`** value. The body is enclosed in curly braces **`{ }`**. If the test fails (results in a **`false`** value) the body will be skipped and the next statement after the body will be executed as shown diagrammatically in Figure 10.1.

In Code Example 10.1 we see an example of an **`if`** statement starting on line 05 and ending on line 07. The test compares the string the user entered with the string **`"hi"`**. If they are the same, a **`true`** value results and the body will be executed.

```
01 <html>
02     <head>
03         <script type="text/javascript">
04             var input = "";
05
06             input = prompt("Enter a string");
07             if(input == "hi") {
08                 alert("Well hello to you too.");
09             }
10         </script>
11     </head>
12     <body>
13         if example
       </body>
   </html>
```

Code Example 10.1: Example using `if`

Exercise 10.1

Using your template create a program that will prompt the user for a number. Convert the user's input to an integer using **`parseInt()`** and store in a variable. Using an **`if`** statement test the input; if the value is greater or equal to zero, output the message "Number was positive".

10.2. The `if-else` Statement

The if-else statement is similar to if but provides a second body which is executed when the test fails.

```
if( TEST ) {
      BODY
}
else {
      BODY
}
```

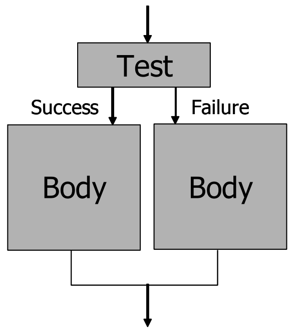


Figure 10.2: How `if-else` works

Only one body is executed as shown in Figure 10.2. After the appropriate body of statements is executed, there is a jump to the next statement after the **`if-else`** statement.

```
<html>
     <head>
          <script type="text/javascript">
               var input = "";

               input = prompt("Enter a string");
               if(input == "hi") {
                    alert("Well hello to you too.");
               }
               else {
                    alert("You entered: "+input);
               }
          </script>
     </head>
     <body>
          if-else example
     </body>
</html>
```

Code Example 10.2: Example using `if-else`

Exercise 10.2

Change the program you created for Exercise 10.1 so that if a user enters a negative number, the message "Number is negative" will be displayed.

10.3. Indenting and Formatting

In programming indenting is used to visually display structure in a program. Indenting is not required for the program to work and has no effect on how the program is executed. However it is good programming practice to use indenting so code is easily readable by humans.

The key to know where to use indenting usually lies in where curly braces **`{ }`** are placed. The content enclosed in braces should be indented one level further than the surrounding code.

```
<html>
     <head>
          <script type="text/javascript">
               var input = "";
               var number = 0;

               input = prompt("Enter a string");
               number = parseInt(prompt("Enter a number"));
               if(input == "hi") {
                    if(number > 0) {
                         alert("You are a positive person");
                    }
                    else {
                         alert("You entered: " + number);
                    }
               }
          </script>
     </head>
     <body>
          Indenting example
     </body>
</html>
```

Code Example 10.3: How indenting is used to show the structure of a program

In Code Example 10.3 an **`if`** statement is used and inside this is another **`if`** statement. The content of the outer (first) **`if`** is indented one level. Within the inner (second) **`if-else`** statement the bodies of the **`if`** and **`else`** are indented again.

Exercise 10.3

Answer the following on paper. What will happen when a user enters:

a. A string other than "hi"?

b. The string "hi" and a number greater than zero?

c. The string "hi" and a number less than zero?

d. The string "hi" and the number zero?

10.4. "Dangling `else`"

Code without indenting is harder to read. In Code Example 10.4 two **`if`** statements are shown without curly braces or indenting. This code achieves the same result as the previous example but is harder to read.

```
<html>
     <head>
          <script type="text/javascript">
               var input = "";
               var number = 0;

               input = prompt("Enter a string");
               number = parseInt(prompt("Enter a number"));
               if(input == "hi")
               if(number > 0)
               alert("You are a positive person");
               else
               alert("You entered: " + number);
          </script>
     </head>
     <body>
          Dangling else example
     </body>
</html>
```

Code Example 10.4: Without indenting code is harder to read

Exercise 10.4

Answer the following on paper.

a. What `if` does the `else` belong to?

b. What would happen `if` a statement was inserted after the second `if` and before the call to `alert()`?

10.5. Guarding Division

Not in implicit curriculum

One application of an `if` statement is to prevent code which could result in unpredictable behaviour or cause the program to crash while being executed. Previously we saw how dividing by zero can produce an unusable result. In some programming languages the effects can be even more severe. It is recommended that you always test the divisor (the second, right-hand operand) before a division operation takes place. If the divisor is zero, division should be avoided.

```
<html>
     <head>
         <script type="text/javascript">
              var number = 0;

              number = parseInt(prompt("Enter a number for division"));
              if(number != 0) {
                   alert(100 / number);
              }
              else {
                   alert("Dividing by zero causes problems");
              }
         </script>
     </head>
     <body>
         Guarding division example
     </body>
</html>
```

Code Example 10.5: The numerator of a division should always be tested before the division

Exercise 10.5

Using your template, create a program that will prompt the user to enter a pre-calculated *sum* of numbers and pre-calculated *count* of numbers. Calculate the *average* (the sum divided by the count). How should your program behave if the user enters zero for the count of numbers?

11. Repetition (Loops)

Often it is desirable to repeat the execution of statements. One way to achieve this is to have the same statements repeated in a program. This can be undesirable because:

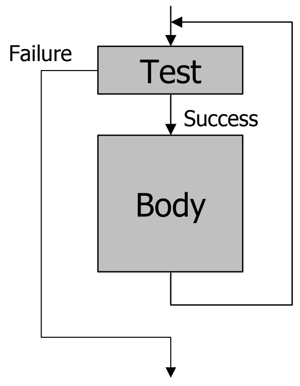
- If a change is needed, each repeated statement will need to be changed. This effort could result in errors.
- It is not possible to achieve a number of repetitions which is determined as the program is running (indefinite repetitions).

A number of structures are provided for achieving repetition.

11.1. `while` Loop

A **`while`** loop works like an **`if`** statement except the body of the loop is executed repeatedly while the test results in a **`true`** value (in other words, until it results in a **`false`** value).

```
while( TEST ) {
      BODY
}
```



When the loop starts, the test is performed; if a **`true`** value results, the body of the loop is executed, otherwise the body is skipped and the next statement after the loop is executed. When the end of the body is reached, the test is run again and this process continues.

```
<html>
     <head>
          <script type="text/javascript">
               var number = 5;

               while(number > 0) {
                    alert("Countdown: "+number);
                    number--;
               }
               alert("BLASTOFF!");
          </script>
     </head>
     <body>
          while loop example
     </body>
</html>
```

Code Example 11.1: Example of a `while` loop

Exercise 11.1

Using your template, create a program that determine if a number is divisible by 2 (**`number%2 == 0`**) and (using **`&&`**) divisible by 3 (**`number%3 == 0`**). If this is the case, output the value and add the number to a sum variable. Repeat this testing within a loop. Start testing at the number 1. Stop looping when the sum is greater or equal to 50. At the end, output the final sum value.

Not in implicit curriculum

11.2. Sentinel Controlled Loops

One application of a while loop is to repeat code until a certain value referred to as a *sentinel* is discovered. Code Example 11.2 shows a poor attempt at achieving a Sentinel Controlled Loop. The program is attempting to count inputs entered by a user before the sentinel is reached.

```
01  <html>
02       <head>
03            <script type="text/javascript">
04                 var input;
05                 var countOfInputs = 0;
06                 var message = "Enter a number (999 to end): ";
07
08                 while(input != 999) {
09                      input = parseInt(prompt(message));
10                      countOfInputs++;
11                 }
12                 alert("Counted "+countOfInputs+" numbers");
13            </script>
14       </head>
15       <body>
16            Bad loop example
17       </body>
18  </html>
```

Code Example 11.2: Repeating until a sentinel is found – this example will produce an incorrect result

This example is deficient because:

- The value of **`input`** is not initialised or set before it is used in the test at line 08. This could have unpredictable consequences.
- The goal of the code is to count numbers before the sentinel is encountered. In this example, when the sentinel is entered by the user it will be included in the count.

A correct solution is shown in Code Example 11.3. In this example, a value for **`input`** is gathered before the test is conducted. If the first number entered is the sentinel, the body of the loop is never executed, which is efficient. On successive inputs the value is always tested before the count is incremented. This will produce the correct answer in the most efficient fashion.

```
01  <html>
02       <head>
03            <script type="text/javascript">
04                 var input = 0;
05                 var countOfInputs = 0;
06                 var message = "Enter a number (999 to end): ";
07
08                 input = parseInt(prompt(message));
09                 while(input != 999) {
10                      countOfInputs++;
11                      input = parseInt(prompt(message));
12                 }
13                 alert("Counted "+countOfInputs+" numbers");
14            </script>
15       </head>
16       <body>
17            Sentinel controlled loop example
18       </body>
19  </html>
```

Code Example 11.3: Repeating until a sentinel is found –allows for the sentinel in the first instance and correctly counts inputs

Exercise 11.2

First plan your solution to the following problem on paper, then implement the program using the template.

Sum floating point numbers collected from a user until they enter 999.

Consider:

a. What variables will be needed? How will they be initialised?

b. How will the loop work? How will the input be collected/converted?

c. When will the output be performed?

11.3. *`for`* Loop

A **`for`** loop is a loop construct with commonly used components conveniently 'built-in'. A **`for`** loop has a number of parts as show in the following syntax description. Note: parts ❶, ❷ and ❹ are separated by semicolons (`;`).

```
for( ❶ INITIALISATION ; ❷ TEST ; ❹ STEP ) {
    ❸ BODY
}
```

The parts of a **`for`** loop are executed in the following order.

❶ INITIALISATION	An opportunity to initialise a counter
❷ TEST	Determines if loop should continue (while **`true`**)
❸ BODY	Actions repeated each time the loop iterates
❹ STEP	An opportunity to move the counter towards the loop end

Note that:

- The initialisation (❶) is only performed once;
- If the test (❷) fails, parts ❸ and ❹ are skipped and the next statement after the loop body is executed; and
- The step (❹) is always followed by the test (❷).

The following code repeats the previous **`while`** loop example using a **`for`** loop.

```
<html>
    <head>
        <script type="text/javascript">
            var counter = 0;

            for(counter = 5; counter > 0; counter--) {
                alert("Countdown: "+counter);
            }
            alert("BLASTOFF!");
        </script>
    </head>
    <body>
        for loop example
    </body>
</html>
```

Code Example 11.4: Example of a `for` loop

Exercise 11.3

Using your template, create a program that will use a *sum* and a *counter*. Set the counter to 1 and loop until it reaches 100 (`counter <= 100`). In each repetition add the value of counter to the sum. At the end output the value of the sum.

Not in implicit curriculum

11.4. Counter Controlled Loops

One application for a `for` loop is to repeat a body of statements a pre-set number of times. Rather than testing each time to see if the last repetition has been reached, a counter is used. For each repetition, the counter is incremented. When the counter reaches a pre-set value, repetition stops. This happens regardless of the content of the loop.

For example, if five values need to be collected, a Counter Controlled Loop could be used to achieve this. The number of repetitions and the termination of the repetition will be controlled by a counter and not by the values collected.

11.5. Finding the Maximum/Minimum

A common task is to find the maximum or minimum in a set of values. The following plan can be used to achieve a search for a maximum.

1. **Initialisation**
   A variable should be used to store the value of the maximum as the search progresses. Only a single variable is needed.. The variable should be set so when the first value is encountered it will become the new maximum. When searching for a maximum, the variable should be initialised to the minimum possible value. For example, if we were searching for a maximum positive integer (numbers zero or greater), the variable should be initialised to zero.
2. **Repetition**
   When searching a set of values of a know size, a Counter Controlled Loop is used. When the set size is unknown, a Sentinel Controlled Loop is used where the sentinel is a special value at the end of the set, or possibly the absence of any more values.
3. **Comparison**
   Each value of the set needs to be compared with the one stored in the variable. If value from the set is the new maximum it should be assigned to the variable.

Exercise 11.4

Using your template, create a program to find the maximum of 5 numbers entered by a user.

Not in implicit curriculum

11.6. Nesting and Merging

When presented with a problem, a series of goals will emerge which need to be achieved in order to solve the problem.

Abutment

Often the goals may need to be achieved in a certain order, in which case *abutment* is used as shown in Section 7. As an example, when searching for a minimum or maximum, a variable used to store the current max/min must be initialised *before* the search can start and the search must be completed before output can take place; this is abutment.

```
1. Initialise maximum variable
2. Search for maximum variable
3. Output maximum variable
```

Nesting

Sometimes sub-goals may need to be achieved to accomplish a greater goal. Sub-goals may be the body of a selection (**`if`** or **`if-else`**) statement or the body of a loop (**`while`** or **`for`**). This sub-goal is *nested* within a greater goal. In the example of finding a maximum or minimum, the comparison between each value in a set and the current max/min must happen within the repetition which gathers each value of the set. The comparison is nested in the repetition.

```
1. Initialise maximum variable
2. Counter Controlled Loop (Search in set of known size)
    a. Input
    b. Test for maximum
3. Output maximum variable
```

Merging

Often two goals can be achieved at the same time; we can *merge* the two goals. Say we were searching a set of a size unknown before the program began. We may want to count how many values are in the set, as well as determining the minimum or maximum. We could gather the same set of inputs twice, but a better solution would be to merge the counting of values with the comparison for a min/max.

```
1. Initialise maximum variable
2. Initialise counter
3. Input (prime loop)
4. Sentinel Controlled Loop (Search in set of unknown size)
    a. Test for maximum
    b. Increment counter
    c. Input
5. Output maximum variable
```

When two plans are merged, the order in which their commonly located parts are performed is usually not important. For instance, when we merge the maximum-search and count plans above, the initialisation of the variables (steps 1 and 2) could be re-ordered without affecting the outcome. Also the steps "Test for maximum" (a) and "Increment counter" (b) could be performed in the opposite order.

Exercise 11.5

Using your template, create a program to allow a user to enter positive integers until the user enters the sentinel 999. Determine the maximum value entered and the count of values (the value 999 will not be included).

12. Arrays

Often it is necessary to store several similar values, for instance:

- 10 numbers entered by a user, or
- The counts of occurrences of each alphabet letter in some text,

...

We could create variables to store each of these values, but a better solution is to store them together in *array*.

12.1. Declaring Arrays

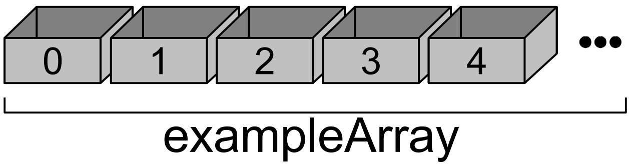
Arrays are declared in a slightly different way to a normal variable.

```
var arrayIdentifier = new Array();
```

...where **`arrayIdentifier`** would be replaced with the identifier for the array, for example...

```
var exampleArray = new Array();
```

...would create an array as follows...



12.2. Accessing Array Elements

Arrays are a collection of individual elements. Once we have created the array we can access each of the elements in an array by using an *index*. Indices are positive integers starting at `0`. The identifier of the array is followed by the index in square brackets `[]`, for example...

```
exampleArray[0]
```

...would allow us to access the first element of **`exampleArray`**. We can assign a value there as follows...

```
exampleArray[0] = 5;
```

...or read a value from that element...

```
alert(exampleArray[0]);
```

In JavaScript arrays are quite flexible.

- Arrays grow as you add to them.
- You can have more than one type of value in the same array.

12.3. Initialising Arrays

It is possible to initialise an array when it is declared. This is done by placing the initial values in the parentheses, with commas in-between each value. The following initialises an array of numbers.

`var monthArray = new Array(31,28,31,30,31,30,31,31,30,31,30,31);`

The following initialises an array of strings.

`var labelArray = new Array("Apples","Oranges","Banannas");`

You can later change the values in the array or add more.

Exercise 12.1

Using your template, declare an array initialised with the names of the days of the week stored as strings. Ask the user to enter a number between 1 and 7 (be sure to convert the input to an integer). Deduct 1 from the input value to get a value between 0 and 6. Use the decremented input as the index to the array and output the day name corresponding to the user's input.

12.4. Arrays for Values

One of the advantages of using arrays is we can perform actions on elements using a loop. Consider the goal of inputting then outputting three numbers. We could create three variables, input values into the three variables, then output the value of each. Alternately we can create an array, we can ask for input in a loop which is repeated three times, then use a loop to output the values of the array (see Code Example 12.1). Now consider what would be required if our goal were extended to 100 numbers. Using variables, this would become quite cumbersome and prone to error. With an array and loops, we merely have to increase the number of repetitions (changing the value of **`numbersToStore`** on line 05 of Code Example 12.1 would achieve this.)

```
01  <html>
02       <head>
03           <script type="text/javascript">
04                var inputArray = new Array();
05                var numbersToStore = 3;
06                var counter;
07                var message = "Please enter a number";
08
09                for(counter=0; counter<numbersToStore; counter++) {
10                    inputArray[counter] = parseInt(prompt(message));
11                }
12
13                for(counter=0; counter<numbersToStore; counter++) {
14                    alert("Input "+(counter+1)+": "+inputArray[counter]);
15                }
16           </script>
17       </head>
18       <body>
19           Array for values example
20       </body>
21  </html>
```

Code Example 12.1: Storing values in individual array elements

Exercise 12.2

Using your template, create a program will allow the user to enter 5 floating point numbers (use **`parseFloat()`**). Store each value in an array and add it to a *sum* at the same time. When input is complete, calculate the average by dividing the sum by 5. For each value in the array output the difference between the average and that value (**`average-numberArray[counter]`**). Some values may be negative and some positive.

12.5. *Arrays for Categories*

Another use for arrays is to count occurrences of items in a set. For instance, we could count "Apples","Oranges" and "Banannas" and store the count of each in an element of an array. The way we could do this is to refer to each item of the set using a number from 0 to 2, say 0 for Apples, 1 for Oranges and 2 for Banannas. We can then use this number as an index to an element in an array.

```
01 <html>
02     <head>
03         <script type="text/javascript">
04             var labelArray = new Array("Apples","Oranges","Banannas");
05             var numFruits = 3;
06             var message = "Please enter:\n";
07             var fruitCountArray = new Array();
08             var counter = 0;
09             var input = 0;
10
11             for(counter=0; counter<numFruits; counter++) {
12                 message = message+counter+" for "+labelArray[counter]+", ";
13             }
14             message = message + "9 to Quit";
15
16             for(counter=0; counter<numFruits; counter++) {
17                 fruitCountArray[counter] = 0;
18             }
19
20             input = parseInt(prompt(message));
21             while(input != 9) {
22                 fruitCountArray[input]++;
23                 input = parseInt(prompt(message));
24             }
25
26             for(counter=0; counter<numFruits; counter++) {
27                 alert(labelArray[counter]+": "+fruitCountArray[counter]);
28             }
29         </script>
30     </head>
31     <body>
32         Array for categories example
33     </body>
34 </html>
```

Code Example 12.2: Using an array to count occurrences of a set of elements

In Code Example 12.2 we count occurrences of fruit. The list of fruit is given in the array **`labelArray`** declared on line 04. We use an array here for the labels reasons:

1. We can declare the labels in one place and refer to them later, and
2. Declaring them in an array gives them order from 0 to 2.

On line 07 we declare the array which we will use to keep a count of the occurrences of each fruit.

On lines 11 to 14 we create a message which we can use later to prompt a user to enter the code number for a particular fruit. We could use the simpler prompt, "Enter the fruit code", but here we are giving the code numbers as well.

On lines 16 to 18, we initialise the count of fruit by setting each array element to zero.

Between lines 20 to 24 is where the action is. The loop will continue until the user enters a 9. On line 22 we see how we are using the code number specified by the user as the index to the array. If the user enters a zero they are referring to Apples so we go to the array element containing the count of Apples (**`fruitCountArray[0]`**) and increment (add one to) the value there. If the user entered 1 or 2, the appropriate **`fruitCountArray`** element would be incremented.

Finally on lines 26 to 28 we output the count of each fruit.

Exercise 12.3

Using your template, create a program will input five integers between 0 and 9. For each input increment the corresponding array element. At the end, output the occurrences of values which were input 1 or more times. For instance, if input was...

6, 2, 4, 2, 2

...output would be...

2: 3

4: 1

6: 1

Not in implicit curriculum

12.6. Counting Values in a Set

Code Example 12.2 contains the biggest JavaScript program we have seen so far. Let's look at this solution in terms of the plans used.

1. **Initialisation**
   Before we can start counting set members we need to initialise the count of each element to zero.
2. **Counter Controlled Loop**
   We know how many elements there are in **`fruitCountArray`**. We will therefore use a counter controlled loop (as opposed to a sentinel controlled loop) to initialise each array element.
3. **Input (twice)**
   We need to input fruit code numbers from a user. We do this once to prime the sentinel controlled loop and again at the end of the loop.
4. **Sentinel Controlled Loop**
   There is no limit to the number of times a user could enter a fruit code number. They could enter several code numbers, they could enter 1, or they could enter none by entering the sentinel (menu option 9) in the first instance. A sentinel controlled loop is therefore used to achieve this repetition.
5. **Set Counting**
   We are not entering the value entered by the user directly into our array. Instead we are using a code number that relates to an element in a set (the set of fruit). We are keeping a count of each fruit set member in an element of an array. For convenience we have made use of fruit code numbers (0 to 2) that are equivalent to the indices of the relevant array elements. We can therefore access the appropriate array element by using the value entered by the user. We can then increment the count in that array element using the statement (from line 22)...

```
fruitCountArray[input]++;
```

6. Output

We need to output the counts stored in the **`fruitCountArray`**.

7. Counter Controlled Loop

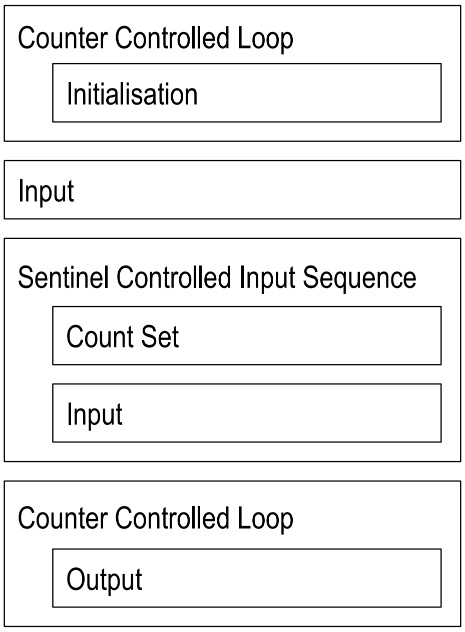
As we know how many elements there are in the **`fruitCountArray`** a counter controlled loop can be used to repeat the output.

Plan Integration

Abutment and nesting can be used to integrate the plans above in a way that will solve the problem.

- Initialisation (1) is nested in the first Counter Controlled Loop (2).
- Set Counting (3) and Input (5) are nested in a Sentinel Controlled Loop (4).
- Output (6) is nested in a Counter Controlled Loop (7).

These plans are abutted in the order (1 to 7) as they appear above.


Counter Controlled Loop
Initialisation
Input
Sentinel Controlled Input Sequence
Count Set
Input
Counter Controlled Loop
Output