

Chapter 7: Flow of control

Introduction

This chapter introduces the concept of **flow of control**. In the Chapter 6 examples, all of the statements within the script are executed one after another. This is known as **sequential flow** and it is illustrated in Figure 7.1, where three statements are shown being executed one after another. If sequential flow was all there was to scripting, then our scripts would be very limited. Luckily, JavaScript enables us to manage the flow of control, to choose which statements are executed and which are not, and also to determine the number of times a set of statements is performed.

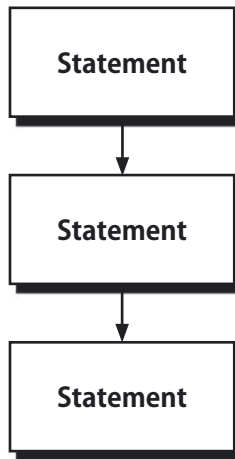


Figure 7.1: Sequential statement processing

The simplest form of flow of control structure is the **if** statement, so we shall examine this first.

if statement

The **if** statement is a conditional statement. It can exist in a number of slightly different forms, each more complex than the previous. We shall begin by examining the **if** statement in its most basic form, as shown below:

```
if (condition)
  statement;
```

Associated with an **if** statement is a condition. The condition is surrounded by parentheses. When encountered, the condition is tested to determine if it evaluates to true. If so, then the statement immediately following the **if** statement is performed. If not true, then this statement is omitted and processing begins with the statement after this one. This is illustrated in the diagram shown in Figure 7.2. Very often programmers indent the statement associated with the **if** statement so that it is clear that the execution of this statement is dependant on the condition within the previous **if** statement being true.

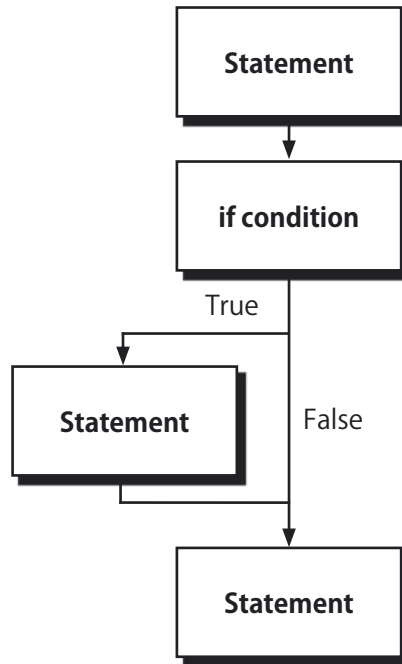


Figure 7.2: if construct

The following script illustrates the use of the **if** statement (albeit in a very unexciting way):

```
1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3 <title>example7-1.htm</title>
4 </head>
5 <body>
6 <script language="JavaScript">
7 <!--
8 var strDay = "Monday";
9 if (strDay == "Monday")
10     document.write("Today is Monday.");
11 //-->
12 </script>
13 </body>
14 </html>
```

In the above script a variable labelled **strDay** is defined on line 8 and initialised to the value **"Monday"**:

```
var strDay = "Monday";
```

On line 10 an **if** statement contains the condition **strDay == "Monday"**:

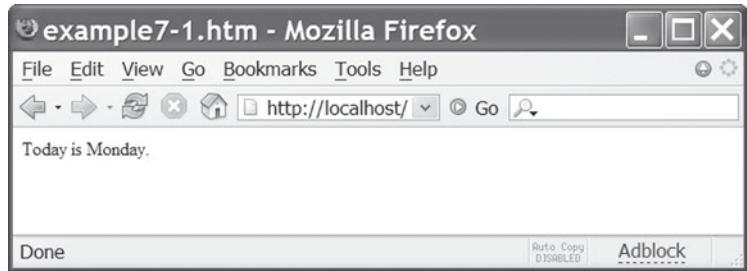
```
if (strDay == "Monday")
```

This condition tests to see if the value contained within the variable **strDay** is equal to **"Monday"**. If so, then the statement on line 11 is executed, otherwise it is not:

```
document.write("Today is Monday.");
```

Figure 7.3 illustrates the output obtained by the script when first run. Try editing the value of variable **strDay** to alter whether the **if** condition evaluates to true or not.

Figure 7.3: Simple if statement output



We mentioned that the **if** construct can appear in a number of different forms. Here is the second, slightly more complex version. In our previous example, the **if** construct only enabled us to control the processing (or not) of a single statement. However, what if we wanted to perform a whole series of statements when the **if** construct evaluated to true? We could do this by using braces to surround the statements:

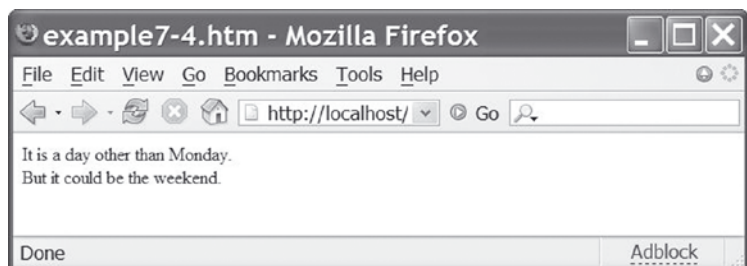
```
if (condition) {
    statements;
}
```

The following script illustrates an example of an **if** construct using braces:

```
1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3 <title>example7-2.htm</title>
4 </head>
5 <body>
6 <script language="JavaScript">
7 <!--
8 var strDay = "Monday";
9 if (strDay == "Monday") {
10     document.write("Today is Monday.<br/>");
11     document.write("What a wonderful start to the week!");
12 }
13 //-->
14 </script>
15 </body>
16 </html>
```

Included within the braces are two **document.write** statements. We could have combined these into a single statement, but that would not have allowed us to illustrate that both statements are performed when the **if** condition is true. The output from the script is shown in Figure 7.4.

Figure 7.4: Conditional if output using braces



Sometimes programmers include braces with **if** constructs even when there is only one statement to be performed. It is argued that this makes for a more consistent style of programming and more readability in the code.



Note

You have to use { } to surround multiple statements that will be executed when the **if** statement evaluates to true. They can be omitted if there is only one statement.

The next enhancement of the **if** construct requires us to introduce a new statement, the **else** statement.

else statement

The **else** statement can be used in conjunction with the **if** statement, but it cannot be used on its own! Remember with our **if** statement examples we were able to execute some statements when the **if** condition was true. Well, the **else** statement allows you to do something when the **if** condition is false. The syntax for the statement is:

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

Figure 7.5 illustrates the **if else** construct and shows that, depending on whether the condition associated with the **if** construct is either true or false, one or another statement will be executed, after which script execution resumes at the next statement within the script.

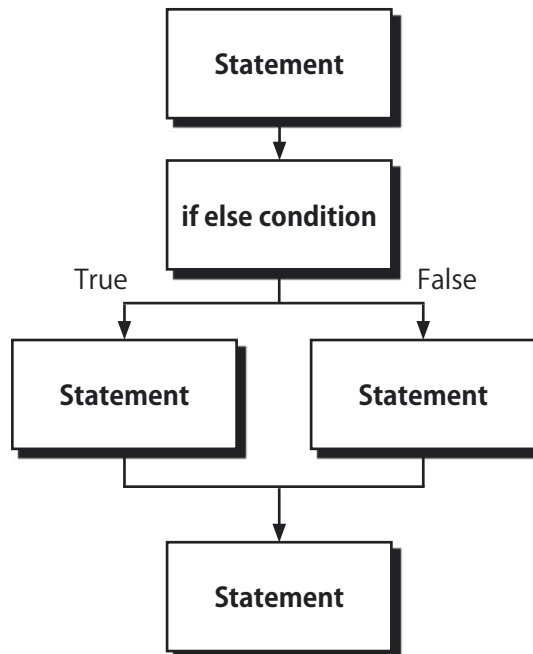


Figure 7.5: if else construct

Chapter 18: Cookies

Introduction

In this chapter we are going to look at the concept of cookies. A cookie is a small amount of data that is associated with a web page. It can be given a name and stored on the client's computer so that when the user returns to the web page or website it can be retrieved and used.

The main purpose of a cookie is to maintain the state of a website or page. Examples of its use are:

- saving and recalling user preferences
- saving and recalling state information
- sharing data between multiple web pages.

We will look at how cookies can be created and retrieved using JavaScript and discuss some of the features and limitations of their use. We will also develop a generalised cookie reading function that can be used whenever you wish to read cookies within your JavaScript code.

First, we will demonstrate how to create a basic cookie.

Writing a simple cookie

Cookies are accessed using the `cookie` property of the document object. It is a string property that enables cookies to be created, written, read, and deleted. The string itself must conform to a specific format and, in addition to the name and value of the cookie, may contain attributes to control the expiry date, path, domain and security of the cookie. More on these later.

Let's now create our first cookie. As we have already stated, a cookie is simply a string that conforms to a specific format. For the basic cookie this string includes the name of the cookie and assigns this to the required value. We then assign this string to the `cookie` property of the document. For example:

```
document.cookie = "name=liz";
```

will create a cookie called **name** and assign it to the value **liz**. Generally we will be assigning the value of a cookie to a JavaScript variable so we would use something like the following:

```
document.cookie = "name=" + strName;
```

Cookie values are not allowed to contain semicolons, commas or white space, so if it is likely the value you wish to store in a cookie contains such characters then you should use a special JavaScript function **escape**, which will replace these characters with a special encoding. For example, if **strName** in the above example was likely to contain spaces then you would use:

```
document.cookie = "name=" + escape(strName);
```

A corresponding function **unescape** exists to decode this string when reading the cookie and return it to its original value. We will see this later when we come to read our cookie.

The following example provides a simple edit box for the user to type their name and a button to call a JavaScript function to create a cookie called **name** and assign it the value entered by the user:

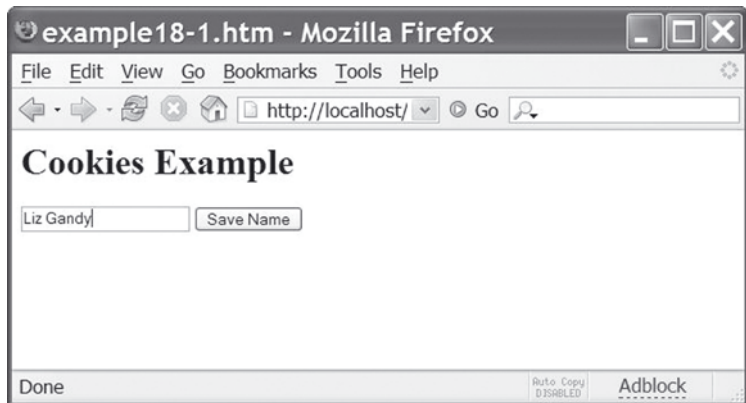
```
1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3 <title>example18-1.htm</title>
4 <script language="JavaScript">
5 <!--
6 function saveName(objForm){
```

```
7     document.cookie = "name=" + escape(objForm.editName.value);
8 }
9 //-->
10 </script>
11 </head>
12 <body>
13 <h1>Cookies Example</h1>
14 <form>
15 <input type="text" id="editName"/>
16 <input type="button" value="Save Name" onclick="saveName(this.form)"/>
17 </form>
18 </body>
19 </html>
```

The function **saveName** creates the cookie by assigning it to a string made up from the cookie name and the contents of the form edit control.

The result of opening this document in the browser is shown in Figure 18.1.

Figure 18.1: Writing a basic cookie



When the user clicks the **Save Name** button, the cookie will be created and given the value **Liz Gandy** but this step will be unseen by the user. Unless they view the source for the document they will be unaware that a cookie has been created and stored on their computer.



Note

The location where cookies are stored on the user's computer depends on the browser being used. Most browsers contain a special cookie directory within their installation.

Having created our first cookie, we will now write a document that retrieves it.

Reading cookies

To read the cookies associated with a particular document we simply assign the cookie property of the document to a string, as in the example below:

```
strCookie = document.cookie;
```

The variable **strCookie** in this case will contain the string that was used to generate the cookie. In fact, it may contain more than one cookie, as it will hold all cookies stored on the user's computer that are accessible by that particular page at that time. We will discuss later how the availability of cookies to different web pages can be controlled. For now, all we need to say is that the cookie created in our first example will be available to any page in the same directory

as this document, or any sub-directory, for the duration of the browser session. If we close the browser down and open it again the cookie will be lost.

The following code provides a very basic facility to read and display the cookie data available to the document when the user clicks a **Read Cookie** button:

```

1 <html xmlns="http://www.w3.org/1999/xhtml">
2 <head>
3 <title>example18-2.htm</title>
4 <script language="JavaScript">
5 <!--
6 function getName(objForm) {
7     var strCookie = document.cookie;
8     document.getElementById("cookieText").innerHTML = unescape(strCookie);
9 }
10 //-->
11 </script>
12 </head>
13 <body>
14 <h1>Cookies Example</h1>
15 <form>
16 <input type="button" value="Read Cookie" onclick="getName(this.form)"/>
17 </form>
18 <span id="cookieText"></span>
19 </body>
20 </html>

```

The cookie is obtained and assigned to a string variable **strCookie**. This string is then written out to the **** place-holder defined on line 18. Notice that, before outputting the cookie string, we call the function **unescape** to decode any semicolons, commas or white space removed from the original value using **escape**:

```
document.getElementById("cookieText").innerHTML = unescape(strCookie);
```

The result of opening this document in the browser (making sure it is still the same browser session that we used to create the cookie) is shown in Figure 18.2.

Figure 18.2: Reading the cookie string



As can be seen, the actual cookie string is output and consists of the cookie name and the value assigned to it, separated by an equals sign. If we wish to obtain the actual cookie value only then we need to write some more complicated JavaScript code into our function to parse the cookie string and pull out the required substring containing our cookie value. We will do this later but, before writing this code, we need to determine what the format of the cookie string looks like if more than one cookie has been stored.