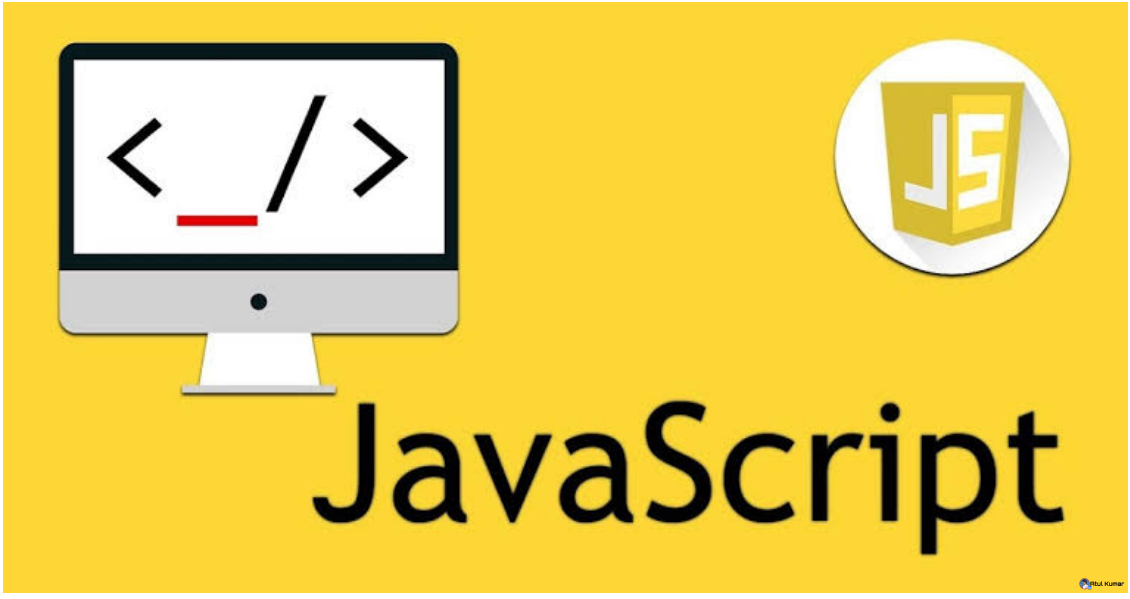
JavaScript for Beginners

Course notes



1 What is a Programming Language?

Key Points

- A programming language is a set of codes that we can use to give a computer instructions to follow.
- Popular and well-known programming languages include Java, C++, COBOL, BASIC, LISP and more. Most popular programming languages consist of words and phrases that are similar in form to the English language.
- A well-written program will be easily readable by anyone with a little programming experience, regardless of whether they have any direct experience of the language in question. This is because modern programming languages share a large number of common concepts. In particular, they all have a notion of **variables**, **arrays**, **loops**, **conditionals**, and **functions**. We will meet these concepts again in more depth later in the course.
- Traditionally, programming languages have been used to write (for the most part) "stand-alone" applications. Things like Microsoft Word, Mozilla Firefox and Lotus Notes are all examples of such applications. Once installed on a PC, these applications run without necessarily requiring any other software to be installed alongside them.
- Web Applications differ from these traditional applications in many respects, but the most striking is that they all run **inside your web browser**. Examples of popular web applications are things like Google, Hotmail, Flickr, GMail and any of the vast array of "weblogging" systems.

- These applications are also written using programming languages, but as a rule they are built using multiple, interdependent technologies. These technologies are easily (though not completely) broken down into two categories: **server-side** and **client-side**.

2 Server-side vs. Client-side

Key Points

- The World Wide Web is built on a number of different technologies.
- For most users, the web starts and ends with their choice of **web browser**. The browser is said to define the **client-side** of the web, with the browser, the computer it is running on, and the user surfing the web being collectively referred to as **the client**.
- Consider a client who has decided to visit the web site at **www.google.com**. The first thing that happens is that the client will make a request to Google's **web server** for the default page of that web site.
- The **web server** is an application running on a computer owned by Google. Like the client, the server application and the computer on which it runs define the **server-side** of the web, and are collectively referred to as **the server**.
- When the server receives the request from the client for a particular page, its job is to retrieve the page from the computer's files and **serve** it back to the client. In many cases, this operation is a very simple procedure involving little or no work on the part of the server.
- However, using a programming language like PHP, Perl or Java, we can cause the server to either modify the page it finds before it passes it back to the client, or even to generate the page entirely from scratch. This is referred to as a **server-side** application. The page passed back to the client looks (to the client) exactly the same as any other page that has not been modified.

- An example of a **server-side** application might be to insert the current date and time into a page. This would mean that each time the page was requested (say, by using the browser's refresh button), a new time value would be added to the page.

- Once the client has received the page from the server, it displays the page and waits for the user to request another page. As soon as the page reaches this state, it has moved beyond the control of the server. No **server-side** application can now alter the contents of the page without the client having to make another trip back to the server to get a new (and possibly updated) copy of the page.

- However, all modern browsers allow for the running of **client-side** applications. These are small applications which are **embedded** within the HTML code of the page itself.

- **Server-side** applications ignore any **client-side** applications that they find while modifying pages to send to the client, so in general the two types of application cannot easily "talk" to each other.

- However, once the client has received a **client-side** application, it can begin to modify the page **dynamically**, without the need to go back to the server.

- An example of a **client-side** application might be a clock on a web page that updated every second.

- An unfortunate side effect of **client-side** applications is that all the code must be sent to the client for running, which means that the application's inner workings are available for anyone to see. This makes it impractical for checking passwords, or doing anything else that could cause confidential information to be released into the wild.

- In addition, all modern web browsers afford the user the opportunity to switch off **client-side** applications altogether. On top of this, the way the same **client-side** application is run will vary from browser type to browser type.

- Despite these drawbacks, **client-side** applications (or **scripts**, as they are better known due to their general brevity) remain the best way to provide web users with a rich environment when developing web applications.

- In short, the two technologies each have their strengths and weaknesses:

- **Client-side** scripts allow the developer to alter pages dynamically, and to respond to user actions immediately rather than having to wait for the server to create a new version of the page. However, there are security and **cross-browser compatibility** issues to be aware of, and these are often non-trivial.

- **Server-side** applications allow the developer to keep her code secure and secret, thus allowing for more powerful applications to be created. In addition, since the server running the code is always a known quantity, applications that run successfully in one browser will run successfully in all browsers. However, despite all this power, there is no direct way for a **server-side** application to alter a page without having to force the **client-side** to load another page. This makes it completely impractical for things like drop-down menus, pre-submission form checking, timers, warning alerts and so forth.

3 About JavaScript

Key Points

- **JavaScript** is an interpreted, client-side, event-based, object-oriented scripting language that you can use to add dynamic interactivity to your web pages.

- **JavaScript** scripts are written in plain text, like HTML, XML, Java, PHP and just about any other modern computer code. In this code, we will use **Windows NotePad** to create and edit our **JavaScript** code, but there are a large number of alternatives available. **NotePad** is chosen to demonstrate **JavaScript's** immediacy and simplicity.

- You can use **JavaScript** to achieve any of the following:

  - Create special effects with images that give the impression that a button is either highlighted or depressed whenever the mouse pointer is hovered over it.

  - Validate information that users enter into your web forms

  - Open pages in new windows, and customise the appearance of those new windows.

  - Detect the capabilities of the user's browser and alter your page's content appropriately.

  - Create custom pages "on the fly" without the need for a server-side language like PHP.

  - And much more…

- **JavaScript** is *not* **Java**, though if you come from a Java background, you will notice that both languages look similar when written. **Java** is a full featured and comprehensive programming language similar to C or C++, and although **JavaScript** can interact with **Java** web applications, the two should not be confused.

- Different **web browsers** will run your **JavaScript** in different, sometimes incompatible ways. In order to work around this, it is often necessary to use **JavaScript** itself to detect the capabilities of the browser in which it finds itself, and alter its operation depending on the result.

- To revisit the original definition in this chapter, note the following points:

  - **Interpreted** refers to the fact that **JavaScript** code is executed (acted on) as it is loaded into the browser. This is a change of pace from **compiled** languages like Java, which check your program thoroughly before running a single line of code, and can have many implications that can catch you out if you are from a non-interpreted programming background.

  - **Client-side** has been defined already in the previous chapter.

  - **Event-based** refers to **JavaScript's** ability to run certain bits of code only when a specified **event** occurs. An event could be the page being loaded, a form being submitted, a link being clicked, or an image being pointed at by a mouse pointer.

  - **Object-oriented** signals that **JavaScript's** power to exert control over an HTML page is based on manipulating **objects** within that page. If you are familiar with **object-oriented programming**, you will be aware of some of the power that this can bring to the coding environment.

- One final note: While **JavaScript** is a programming language, HTML (the language of the World Wide Web) is *not*. HTML is a **Markup Language**, which means that it can be used to mark areas of a document as having special characteristics like headers, paragraphs, images, forms and so on, but it cannot perform any logical processing on its own. So while **JavaScript** is often written alongside HTML, the rules of one do not necessarily have any bearing on the other.

4 A Tour of JavaScript

Key Points

- Let's start with a quick tour of the major features of **JavaScript**. This chapter is intended to be a showcase of what **JavaScript** can do, not an in depth investigation into the deeper concepts, so don't worry too much if you get lost or don't understand the code you're typing in!

Project

- Our **JavaScript** is all going to be written using **NotePad**. Open **NotePad** and save the resulting empty document in your user drive as **`chapter_4.html`**.

- Begin by creating a basic HTML page in your blank document. It doesn't have to be anything fancy – the following will be more than sufficient:

```
<html>
<head>
    <title>Chapter 4: A Tour of ↲
  JavaScript</title>
</head>


<body>

<h1>A Tour of JavaScript</h1>

</body>
</html>
```

- As a convention, when the notes intend that you should enter code all on one line, they will use an arrow as above ↲ *to indicate that you should not take a new line at that point. With HTML, this is rarely important, but with **JavaScript**, a new line in the wrong place can stop your code from working.*

- Save your new webpage, and view it in your web browser. For the moment, use **Internet Explorer** to view this page. To do this, find your saved file on your user drive, and double-click on it. This will open the file in **Internet Explorer** by default, and let you see the header you've just created.

- So far, we haven't done anything beyond the scope of HTML. Let's add some **JavaScript** to the page.

- There are (generally speaking) **three** places in a web page where we can add **JavaScript**. The first of these is between a new set of HTML tags. These **`script`** tags take the following form:

```
<script language="JavaScript" ↲
  type="text/JavaScript">

… code …

</script>
```

- The **`script`** element above can be placed virtually anywhere you could place any element in an HTML page – in other words, in either the **`head`** element or the **`body`** element. It is most commonly placed in the former, though this is usually so that all your code can be easily found on the page.

- Note too that there is no arbitrary limit on the number of **`script`** elements that can be contained in an HTML page. There is nothing stopping you from having a hundred of these dotted around your pages, except perhaps prudence.

- Let's add our opening and closing script tags to the head element of the page, like so:

```
<html>
<head>
    <title> … </title>
    <script language="JavaScript" ↲
  type="text/JavaScript">

    </script>
</head>

…
```

- Save the file, and then try refreshing your page in the browser window. Note that nothing has happened. This is what we expected – all we have done so far is to set up an area of the page to *hold* our **JavaScript**.

- Go back to **NotePad** and enter the following text between the opening and closing **`script`** tags:

```
window.alert("Hello world!");
```

- Save your changes, and again refresh your page in the browser window. Welcome to the world of **JavaScript**!

- Go back to notepad and remove the **`window.alert`** line you just added. Now add the following, slightly more complex code:

```
if  ( confirm("Go to Google?") )  {
    document.location  =  ↲
"http://www.google.com/";
}
```

- Again, save your changes and refresh the page. For those with an eye to future chapters, this is an example of a **conditional** statement, where we ask **JavaScript** to check the **condition** of something (in this case, our response to a question) and then to alter its behaviour based on what it finds.

- Now, both of these bits of **JavaScript** have run uncontrollably when the page has loaded into the browser. In most cases, we will want to have more control over when our **JavaScript** does what we ask it to.

- This control is the domain of **events**. In a browser, every element of an HTML document has associated with it a number of **events** that can happen to it. Links can be **`click`**ed, forms can be **`submit`**ted, pages can be **`load`**ed and so on.

- Modify the previous lines of JavaScript in your **`script`** element to match the following:

```
function  go_to_google() {
  if  ( confirm("Go to Google?") )  {
    document.location  =  ↲
"http://www.google.com/";
  }
}
```

- Be careful with your brackets here!
- Save and refresh, and note that nothing happens this time. This is because we have enclosed the previous action (popping up a question and acting on the response) within a **function**. A **function** is a block of code that is given a name – in this case, the name is **`go_to_google()`** – and is only run when that name is "called". It can be useful to think of **functions** as magic spells that can be invoked when their name is said.
- To invoke this spell, we need to choose an element on the page to trigger it. A natural candidate is a link element, so add the following HTML to the **`body`** section of your page:

```
<p>A quick <a href="#">test</a>.</p>
```

- The # link is a common HTML trick that allows us to create a "link to nowhere".
- Save and refresh, and check that the link appears on the page, and that it goes nowhere when clicked.
- Now, we want to have our page ask us if we want to "Go to Google?" when we click on that link. Here's how
- Take the link element, and modify it as follows:

```
<a href="#" onclick="go_to_google();">test</a>
```

- Save and refresh, and then click on the link. This is an example of an **event handler**. When the link is clicked (**`onclick`**), our browser says the "magic words" **`go_to_google()`**, and our **function** is invoked.
- For our final trick, add the following code to the **`body`** section of the page, after the paragraph containing the link:

```
<body>
…
<script language="JavaScript" ↲
  type="text/JavaScript">

document.write("<h2>Here's another ↲
  header!</h2>");

</script>
```

- Note that the line of code should be all on one line!

- Save the page and refresh the browser. Note that we now have a new line of text on the page – another header! We've used **JavaScript** to create HTML and tell the browser to display it appropriately. In this example, **JavaScript** has done nothing that we couldn't have done with a line of HTML, but in future chapters we will see how we can use this to write the current date and more.

5 Objects, Properties and Methods

Key Points

- Generally speaking, **objects** are "things". For example, a piano is an **object**.
- **Properties** are terms that can describe and define a particular **object**. Our piano, for example, has a colour, a weight, a height, pedals, a keyboard and a lid.
- Note from the above that an object's properties *can be properties themselves*. So we have the case where a piano lid is a property of the piano, but is also an object in its own right, with its own set of properties – for example, the lid has a colour, a length, and even a *state* of either open or closed.
- If **objects** are the nouns of a programming language and **properties** are the adjectives, then **methods** are the verbs. **Methods** are actions that can be performed on (or by) a particular object. To continue our piano example, you could play a piano, open its lid, or press the sustain pedal.
- Many programming languages have similar ways of referring to objects and their properties or methods. In general, they are *hierarchical*, and an object's relationship with its properties and methods, as well as with other objects, can often be easily seen from the programming notation.
- In JavaScript, we use a "dot notation" to represent objects and their properties and methods. For example, we would refer to our piano's colour in the following way:

```
piano.colour;
```

- If we wanted to instruct JavaScript to play the piano, we could write something as simple as:

```
piano.play();
```

- A clear example of object hierarchy could be seen if we decided to open the lid of the piano:

```
piano.lid.open();
```

- Or even more so if we wanted to press the sustain pedal of the piano:

```
piano.pedals.sustain.press();
```

- Note that in some of the examples above, we have brackets () after each set of words, and in some we don't. This has to do with making sure that JavaScript can understand what we say.

- JavaScript works with objects throughout its existence in a web browser. All HTML elements on a page can be described as objects, properties or methods. We have already seen a few of these objects in our previous introductory chapter:

```
document.write(…);
document.location;
```

- In these examples, **`document`** is an object, while **`write`** is a method and **`location`** is a property.

- In these lines, we see a clue about the use of brackets in these statements. We use brackets to signify to JavaScript that we are talking about an object's **method**, and not a property of the same name.

- Brackets also allow us to pass certain extra information to an object's method. In the above example, to write the text "Hello world!" to a web page document, we would write the following JavaScript:

```
document.write("Hello World");
```

- Each method can do different things depending on what is put in the brackets (or "passed to the method as an argument", to use the technical term). Indeed, many methods can take multiple "arguments" to modify its behaviour. Multiple arguments are separated by a comma (,).

- A JavaScript instruction like those shown here is referred to as a JavaScript **statement**. All statements should end in a single semi-colon (;). JavaScript will often ignore missed semi-colons at the end of lines, and insert the semi-colon for you. However, this can cause some unexpected results. Consider the following:

```
document.write("<h1>
                    Hello World!
                </h1>");
```

- In many other languages, this would be acceptable. However, JavaScript will often interpret something like this as the following:

```
document.write("<h1>;
Hello World!;
</h1>");
```

- This interpretation will generate an error, as JavaScript will complain if you end a statement without ensuring that any terms between quotes have matching pairs of quotes. In this example, the first line's "statement" is cut short, and JavaScript will fall over.

- For this reason, it is recommended that all your statements should end with semi-colons.

6 Assigning Values to Properties

Key Points

- While objects and methods allow us to **do** things on a page, such as alter the content or pop up dialogue boxes to interact with the user, in many cases we will want to alter the value of one of an object's properties directly. These cases are akin to painting our piano green.

- Given our discussion on methods so far, we might expect to be able to alter our object's properties by using a method – for example, the following would seem logical:

```
piano.paint("green");
```

- In many cases, that is exactly what we will do. However, there are two drawbacks here. The first is that, within this course, the majority of objects that we discover are built into and defined by our browser. If we rely on using a method to alter an object's property, we are also relying on the fact that the method exists in the first place.

- A much more direct way to solve this problem is to access the object's properties directly. For example:

```
piano.colour = "green";
```

- Here we are no longer using a method to perform an action, we are using what is known as an **operator**. In this case, the operator has the symbol "=", and is known as the **assignment operator**.

- Within JavaScript, we can use this operator to great effectiveness. For example, we could alter the title element of a document (the text that is displayed in the top bar of the browser's window) dynamically. This could be done when a user clicked on a part of the page using an event handler (more later on this), or could be set to automatically update each minute to show the current time in the page title. The code we would use for this task is simple:

```
document.title  =  "a new title";
```

- There are many assignment operators in JavaScript. Some of the more common are shown in the table below:

Assignment	Function
x = y	Sets the value of x to y
x += y	Sets the value of x to x+y
x -= y	Sets the value of x to x-y
x *=y	Sets the value of x to x times y
x /=y	Sets the value of x to x divided by y

- Not all assignment operators work with all types of values. But the addition assignment operator works with both numbers and text. When dealing with numbers, the result will be the sum of the two numbers. When dealing with text (technically called **strings**), the result will be the **concatenation** of the two strings:

```
document.title += "!";
```

will cause the symbol "!" to be appended to the end of the current document title.

Project

- Open your previous project file, and save it under the name **`chapter_6.html`**.

- Remove any existing JavaScript from your script tags, but leave the tags in place ready for some new JavaScript.

- Use your text editor to change the value of the title element of the page as follows, then load your page into a browser and view the result:

```
<title>With a little help from</title>
```

- Now, add a statement to our script element to add the following text to the end of the current title:

```
"JavaScript for Beginners!";
```

- Reload the page in your browser and note the title bar of the window.

- If the display looks odd, consider your use of spaces…

- All we have so far is an example that does nothing more than HTML could manage. Let's introduce a new method of the **window** object to help us to add a little more dynamism and interaction to the script. Change the value of the title tag as follows:

```
<title>Chapter 6: Assigning Values to
Properties</title>
```

- Now, remove your previous JavaScript statement and insert the following:

```
document.title = ↲
   window.prompt("Your title?", "");
```

- Reload your page and consider the result.

- We have come across the **window** object before. Our demonstration of the **alert** method in chapter 4 could have been more properly written as:

```
window.alert("message");
```

  In many cases, we can omit certain parts of our object/property/method hierarchy when writing our code. We will discuss this again later.

- To understand what is going on with our **prompt** method, we can write down a method **prototype**. This is a way of describing a method's arguments in such a way that their effect on the method is more self explanatory. A prototype for the prompt method of the window object might look like the following:

  ```
  window.prompt( message, default_response );
  ```

- So, we can see that the first argument defines the text that appears as the question in the prompt dialogue box. The second argument is a little less clear. Try your code with different values and see what difference your changes make.

- Finally, we note that this prompt method somehow takes the information typed into the box and passes it to our JavaScript assignment. Say someone typed "Hello World" into the box. It would have been as if our assignment had actually been:

  ```
  document.title = "Hello World";
  ```

- When this sort of passing of values occurs, it is said that the method has **returned** the value passed. In this case, we would say that "the prompt method has returned the value 'Hello World'", or that "the return value of the prompt method was 'Hello World'".

- Return values will become very important when we deal with event handlers later on.

7 About Comments

Key Points

- Repeat after me : Comments are important. Comments are **important**. **Comments are important**.
- Adding comments to your code is always good practice. As the complexity of your scripts grows, comments help you (and others) understand their structure when you come to view the code at a later date.
- A lot of code created quickly is said to be "write only" code, as it suffers from an inherent lack of structure or commenting. Debugging such code, or reusing it months later, becomes maddeningly impossible as you try to remember what a certain line was supposed to do, or why using certain values seems to stop your code from working.
- Comments are completely ignored by JavaScript and have no effect on the speed at which your scripts run, provided they are properly formed.
- Comments *can* slow the loading of your page, however – many coders keep a "development" copy of their code fully commented for editing, and remove all comments from their code when they finally publish it.
- There are two types of comment in JavaScript – single line comments, and multi-line comments.
- Single line comments begin with two forward-slash characters (//), and end with a new line:

```
// this is a comment

alert("hello"); // so is this
```

- Single line comments in JavaScript can also use the HTML comment format that you may be familiar with:

```
<!-- this is a comment

alert("hello");
```

- Note two things: firstly, this use of the HTML comment format **does not** require a closing **-->** tag. Secondly, this is only a one line comment, unlike its use in HTML, which comments all lines until the closing comment tag.

- You can add multiple-line comments by enclosing the comment block between /* and */. For example:

```
/* all of this text is going to be
ignored by JavaScript. This allows us to
write larger comments without worrying about
having to individually "comment out" each
line */

alert("Hello World");

/* a one line, "mult-line" comment */
```

- As well as adding narrative to your script, you can use comments to remove code from your pages without having to delete the code. For example:

```
// this was the old message
// alert("Hello World");
// and this is the new message
alert("Hello everyone!");
```

- This can be very useful if you are trying to track down an error in your code – you can "comment out" each suspect line in turn until you manage to get your code working again.

Project

- Open your previous project file, and save it under the name **chapter_7.html**.

- Add the single line comment

```
This is my first comment
```

to the beginning of your script.

- Add a multi-line comment to your script, replacing your previous single line comment. The multi-line comment should describe what your script does at present.

8 Hiding Scripts from Older Browsers

Key Points

- Very old browsers don't understand JavaScript. There are very few such browsers in use today, but two factors force us to continue to consider environments that may not be able to cope with our JavaScript code.

- Firstly, all modern browsers allow users to control whether JavaScript code will be run. In many cases, users will not have any say over their company policy, and may not even know that their work machine has had JavaScript disabled.

- Secondly, not all of your visitors will be using browsers that can make any use of JavaScript. Braille displays, screen readers and other non-visual browsers have little use for many JavaScript tricks. In addition, search engines like Google will ignore any JavaScript you use on your pages, potentially hiding important content and causing your pages to remain un-indexed.

- Browsers that don't support JavaScript are supposed to ignore anything between the opening and closing script tags. However, many break this rule and will attempt to render your code as HTML, with potentially embarrassing consequences.

- However, we can use the fact that **<!--** denotes a single line comment in JavaScript but a multi-line comment in HTML to ensure that our code is seen by a JavaScript-savvy browser, but ignored as commented-out HTML by anything else:

```
<script>
<!-- hide  from older browsers

… your code

// stop hiding code -->
</script>
```

- This prevents older browsers from displaying the code, but what if we want to replace this with some comment. For example, let's say we had a bit of code that displayed the time of day and greeted our user by name. Without JavaScript and using the method above, there would simply be a blank on the page where the greeting should have been.

- We can use the <noscript> tag to cause our code to "fail gracefully" where JavaScript has been disabled or is unavailable. The contents of this element will be ignored where JavaScript is understood, and displayed anywhere else. For example:

```
<noscript>
    <h1>Welcome to our site!</h1>
</noscript>

<script>
<!-- hide  from older browsers

… code to customise header

// stop hiding code -->
</script>
```

Project

- Open your previous project file, and save it under the name **chapter_8.html**.

- Add two lines to your code to ensure that it will not confuse older browsers or browsers where the user has disabled JavaScript.

- Add a noscript element to explain what your JavaScript does. It is generally considered “bad form” to instruct your user to “upgrade to a better browser”, as this can insult many people who use assistive devices – consider this form of advice to be similar to the advice that tells a blind person “to get some glasses”.

- Instead where possible you should use the noscript element to provide content that adequately replaces the scripted content with a suitable replacement. For example, if you use your JavaScript to build a navigation panel on your page, the noscript element should contain a plain HTML list that does the same job.

9 Automatically Redirecting the User

Key Points

- We have already briefly seen the use of browser redirection in chapter 4.
- To formulate the idea more completely, in order to redirect the user to a different page, you set the **location** property of the **document** objects.
- As we saw in chapter 6, we can use the assignment operator here. For example:

```
document.location = "http://www.bbc.co.uk/";
document.location = "chapter_4.html";
```

Project

- Open your previous project file, and save it under the name **`chapter_9_redirect.html`**.
- Save another copy of the file, this time called **`chapter_9.html`**.
- Make sure both files are saved in the same folder, and that you have **`chapter_9.html`** open in your editor.
- Remove all script from between the script tags, except for your browser hiding lines. Make sure that the script tags are still in the head section of the page.
- Now, add a single statement to this script that will automatically redirect the user to the page **`chapter_9_redirect.html`** as soon as the page is loaded into a browser.

- Finally, add a header tag to the body section of the page containing the text “You can’t see me!”.
- Close this page, don’t check it in a browser yet, and open the page **`chapter_9_redirect.html`** in your editor.
- Remove all JavaScript from this page (including your script tags) and ensure that only HTML remains on the page.
- Add a header tag to the body section of the page containing the text “But you can see ME!”.
- Save this page, and load the page **`chapter_9.html`** into your browser.
- Experiment with various positions for the script tags on **`chapter_9.html`** to see if you can make the header appear before the redirection.

10 Alert, Prompt and Confirm

Key Points

- So far, we have seen brief examples of **alert**, **prompt** and **confirm** dialogue boxes to request a response from the user, and to pause all code in the background until the request is satisfied.

- All of these boxes are the result of methods of the **window** object. This object is the highest level object that JavaScript can deal with in a browser. As such, all other objects on a page (with a few exceptions) are actually properties of the window object.

- Because of this ubiquity, its presence is assumed even if it is omitted. Thus, where we technically *should* write:

  ```
  window.document.write("…");
  ```

  it is equally valid to write:

  ```
  document.write("…");
  ```

  as we have been doing.

- Similarly, instead of writing:

  ```
  window.alert("…");
  ```

  we can happily write:

  ```
  alert("…");
  ```

- The prototypes of the three methods are:

```
window.alert( message );
window.confirm( message );
window.prompt( message, default_response );
```

- Alert will always return a value of "true" when it is cleared by clicking "ok".
- Confirm will return either "true" or "false" depending on the response chosen to clear the box.
- Prompt will return either the value typed in, "null" if nothing is typed in, and "false" if the box is cancelled.

Project

- Open your previous project file, and save it under the name **`chapter_10.html`**.
- Clear the previous redirection code, and ensure that the script tags have been returned to the head section of the document.
- Add a new statement to the script on the page that will display the following message before the rest of the page is shown:

```
Welcome to my website! Click OK to continue.
```

- Check your page in your browser.
- We will use **alert**, **confirm**, and **prompt** throughout this course. Take a moment to try each of them in turn on this page, each time stopping to review your changes.
- Use the write method of the **document** object to check the return values of each method. For example:

```
document.write(alert("hello world"));
```

Make sure that you place this particular snippet of code in script tags within the body area of the page, as we are generating text output to be rendered by the browser. Also, note the use (or not) of quotes here. More next chapter!

11 Variables and Operators

Key Points

- We have been introduced to the concepts of **objects** and their various **properties** and **methods**. These inter-related concepts allow any web page to be broken down into little snippets of information or **data**, which can then be accessed by JavaScript and, in many cases, changed.

- However, what if we want to create our own storage space for information that doesn't necessarily have a page-based counterpart? For example, what if we wanted to store the previous value of a document's title property before changing it, so it could be retrieved later, or if we wished to store the date time that the page was loaded into the browser for reproduction in several places on the page, and didn't want to have to recalculate the time on each occasion?

- **Variables** are named containers for **values** within JavaScript. They are similar to object properties in many ways, but differ importantly:

- In a practical sense, variables have no "parent" object with which they are associated.

- Variables can be created ("declared") by you as a developer, and can be given any arbitrary name (within certain rules) – object properties, however, are restricted by the definition of the parent object. It would make no sense, for example, for our piano object in the previous chapters to have a propeller property!

- Variable name rules are straightforward – no spaces, names must start with a letter. Examples of valid variable names are:

  BrowserName
  page_name
  Message1
  MESSAGE1

- In many browsers, JavaScript is **case-sensitive**, which means that the last two variables in the example above are **distinct variables**. It is a good idea to pick a particular naming style for your variables, and to stick to it within your projects.

- At the simplest level, variables can store three different types of value:

- **Numbers**
  e.g. 1.3324, 3.14159, 100000, -8 etc.

- **Strings**
  e.g. “JavaScript for Beginners, week 3”, “Hello World” etc.

- **Boolean Values**
  e.g. true, false

- Note that strings can contain numbers, but the following variable values are **not** equivalent:

  1.234 and “1.234”

  The latter is a **string value**. This becomes important. Consider:

  1+2 = 3
  “a” + “b” = “ab”
  “1” + “2” = “12”

- Some developers use their own naming convention with variable names to denote the type of value expected to be contained in a given variable. This can often be helpful, but is in no way required by JavaScript (c.f. JavaScript comments)

- For example, **strMessage** might indicate a string variable, where **numPageHits** might indicate a numerical value in the variable.

- **Variable assignment** is accomplished in the same way as object property assignment. When a variable is assigned a value for the first time, it is automatically created. This is different from other programming languages, where a variable must be created explicitly first, before it can be loaded with a value.

- Some examples of variable assignment follow:

```
numBrowserVersion = 5.5;

numTotal += 33;

Message = "Hello!";
Message = "Goodbye";
Message = 3;
```

- Note that the last three examples show that variable values can be altered after their initial assignment, and also that the type of value stored in a variable can be altered in a similar manner.

- Once a variable has been created and a value stored within, we will want to be able to access it and perhaps manipulate it. In a similar manner to object properties, we access our variables simply by **calling them:**

```
Message = "Hello World!";
alert(Message);
```

- Note that we do not use quote marks around our variable names. The above code is different from:

```
alert("Message");
```

  for hopefully obvious reasons.

- As well as using variables for storage and access, we can combine and manipulate them using **operators**. For example:

```
a = 12;
b = 13;
c = a + b; // c is now 25
c += a; // c is now 37
c = b + " Hello!"; // c is now "13 Hello!"
```

- Our last example may have been unexpected – we added a number to a string and got a string as a result. JavaScript is smart enough to realise that a number cannot be "added" to a string in a numerical sense, so it converts the number temporarily to a string and performs a **concatenation** of the two strings. Note that **b** remains **13**, not **"13"**.

- A table of operators:

Operator	Function
x + y	Adds x to y if both are numerical – otherwise performs concatenation
x – y	Subtracts x from y if both are numerical
x * y	Multiplies x and y
x / y	Divides x by y
x % y	Divides x by y, and returns the remainder
-x	Reverses the sign of x
x++	Adds 1 to x AFTER any associated assignment
++x	Adds 1 to x BEFORE any associated assignment
x--	Subtracts 1 from x AFTER any associated assignment
--x	Subtracts 1 from x BEFORE any associated assignment

Project

- Open your previous project file, and save it under the name **`chapter_11.html`**.

- Clear the previous JavaScript code, and ensure that the script tags are contained in the body section of the document.

- Assign the message

  `"Welcome to my web site"`

  to a variable called **`greeting`**.

- Use this variable to create an alert box containing the message, and also to produce a header on the page without having to retype the message.
- Test this page in your browser.
- Now, modify your code to create two variables, **`var_1`** and **`var_2`**.
- Assign the value “Welcome to” to **`var_1`**, and the value “my web site” to **`var_2`**.
- Create a third variable **`var_3`**, and assign to it the value of **`var_1 + var_2`**. Then use an alert box to check the resultant value of **`var_3`**.
- Test this page in your browser.
- If the text in the alert box does not appear as expected, consider the use of spaces in the variable assignments, and correct the error.
- Now, modify your code to produce the same result but without requiring a third variable.
- Clear all statements from the current script tags.
- Add two statements to the script which assign the **numbers** **`100`** to one variable and **`5.5`** to another.
- Use **`document.write`** to show the effects of each of the operators given in the table on page 34 on the two numerical values.
- Substitute one of the numerical values for a text string and repeat the procedure. Note the differences.

12 Comparisons

Key Points

- Comparison operators compare two values with each other. Most commonly, they are used to compare the contents of two variables – for example we might want to check if the value of **`var_1`** was numerically greater than that of **`var_2`**.

- When you use a comparison operator, the value that is **returned** from the comparison is invariably a **Boolean** value of either **`true`** or **`false`**. For example, consider the following statements:

```
var_1 = 4;
var_2 = 10;

var_3 = var_1 > var_2;
```

  In this case, the value of **`var_3`** is false. Note that the **Boolean** value of false is not the same as the text string **`"false"`**:

```
var_4 = false;  // Boolean value
var_5 = "false";  // Text string
```

- Common comparison operators are given below:

Comparison	Function
X == y	Returns true if x and y are equivalent, false otherwise
X != y	Returns true if x and y are **not** equivalent, false otherwise
X > y	Returns true if x is numerically greater than y, false otherwise

X >= y	Returns true if x is numerically greater than or equal to y, false otherwise
X < y	Returns true if y is numerically greater than x, false otherwise
X <= y	Returns true if y is numerically greater than or equal to x, false otherwise

- To reverse the value returned from a comparison, we generally modify the comparison operator with a **!** (a "bang"). Note that in many cases this is not necessary, but can aid comprehension:

```
var_1 !> var_2;
var_1 <= var_2;
```

both of these are equivalent, but one may make more semantic sense in a given context than the other.

Project

- Open your previous project file, and save it under the name **chapter_12.html**.
- Ensure that your two variables both have numerical values in them and not strings.
- Use an alert box to display the result of a comparison of your two variables for each of the comparison operators listed above.
- Substitute one of the numerical values for a text string and repeat the procedure. Note the differences.

13 Conditionals

Key Points

- Up until now, our JavaScript projects have been unable to alter their behaviour spontaneously. When a page loads with our JavaScript embedded within, it is unable to do anything other than what we expect, time and again.
- The only difference we have seen is in the use of a prompt box to alter what is shown on a page. However, the page essentially does the same thing with the text provided, regardless of what text is typed in.
- What would be really handy would be to give JavaScript a mechanism to make decisions. For example, if we provided a prompt box asking the visitor for their name, it might be nice to have a list of "known names" that could be greeted differently from any other visitors to the site.
- **Conditional statements** give us that ability, and are key to working with JavaScript.
- A conditional statement consists of three parts:
- A test (often with a comparison operator, or **comparator**) to see **if** a given condition is **true** or **false**.
- A block of code that is performed if and only if the condition is **true**.
- An optional block of code that is performed if and only if the condition is **false**.

- These three parts are represented in JavaScript as follows:

```
if ( conditional_test )
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
else
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
```

- Everything from the first closing curly bracket (or **brace**) is optional, so the following is also a valid conditional prototype:

```
if ( conditional_test )
   {
   JavaScript statement;
   JavaScript statement;
   JavaScript statement;
   …
   }
```

- In this case, if the **`conditional_test`** does not return **true**, nothing happens.

- An example of complete conditional statement is as follows:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
   {
   alert("Variable 2 is greater");
   }
```

- Note that the above condition is not necessarily always correct. Consider the case where **var_1** is equal to **var_2**. In that case, the above code will produce the message that "Variable 2 is greater", since **var_1 > var_2** returns **false**. In this case, we want to add an additional condition to the **else** branch of code:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
if ( var_1 < var_2 )
   {
   alert("Variable 2 is greater");
   }
```

- In this case, equality will produce no output, as neither of the conditions will return true. For completeness, we could add a final else branch to the statement:

```
if ( var_1 > var_2 )
   {
   alert("Variable 1 is greater");
   }
else
if ( var_1 < var_2 )
   {
   alert("Variable 2 is greater");
   }
else
   {
   alert("The variables are equal");
   }
```

- Note that in this case, we don't have to check for equality in the final branch, as if **var_1** is neither greater than nor less than **var_2**, then – numerically at least – the two must be equal.

- We can continue adding as many else if statements as required to this stack.

- If you only have one statement following your conditional test, the braces may be omitted:

```
if ( var_1 > var_2 )
   alert("Variable 2 is greater");
```

  However, if you later want to add further statements to this conditional branch, you will have to add braces around the block, and this can lead to confusion. It is recommended that you use braces to enclose all blocks of conditional code.

- Consider the following block of code:

```
if ( var_1 > 4 )
   {
   var_2 = var_1;
   }
else
   {
   var_2 = 4;
   }
```

- This code is rather long, but achieves comparatively little – **`var_2`** is equal to **`var_1`** or **`4`**, whichever is greater.

- A more compact way of writing this could be:

```
var_2 = 4;
if ( var_1 > var_2 )
   {
   var_2 = var_1;
   }
```

- However, an even more compact way of writing this could be to use the **ternary operator**:

```
var_2 = (var_1 > 4) ? var_1 : 4;
```

- In the above statement, the conditional is evaluated and, if true, the value returned is the value between **`?`** and **`:`** symbols, or if false, it is the value between the **`:`** and **`;`** symbols.

Project

- Open your previous project file, and save it under the name **`chapter_13.html`**.

- Clear all JavaScript code from your script tags.

- Create two variables and assign numerical values to them.

- Use a conditional statement to show alert boxes declaring which variable is the greater of the two.
- Consider the following code:

```
var_3 = (var_1 > var_2);
```

- Use this code in your script to simplify your conditional checking code.
- Now, use a prompt box to ask your visitor their name. Assign this name to **`var_3`**.
- Check to see if the name typed in is your own name. If it is, use **`document.write`** to display a personalised greeting on the page. Otherwise, display a generic greeting.
- Use multiple else if branches to check the typed name against the names of some of your friends. Create personalised messages for all of them.
- There may be a way to simplify your conditional greeting code to use only one **`document.write`** statement. See if you can figure out how. Hint – how might you use a variable called **`greeting`**?

Project 2

- In many cases, the brevity of your conditional statements will rely on your ability to formulate the right "questions" to consider when performing your tests. Try to make your solution to the following problem as concise as possible.
- Clear all of your current code from the script tags.
- Ensure that your script tags are currently situated in the body section of the page.
- Create a variable called **`exam_result`** and store a numerical value of between **`0`** and **`100`** in it.

- Use an **`if`** statement and multiple **`else if`** statements to check the value of this variable against the following exam grading scheme, and print out the appropriate message to the page:

Exam Result	Result Message
90 or more	Excellent. Pass with Distinction.
Between 70 and 89	Well Done. Pass with Honours
Between 55 and 69	Just passed.
54 or below	Failed. Do better next time.

- Test your result in your browser. Vary the value of **`exam_result`** and check the value shown in the browser. For extra practise, try to use a prompt box to make changes to your **`exam_result`** variable as easy to achieve as possible.

14 Looping

Key Points

- The letters **`i`**, **`j`** and **`k`** are traditionally used by programmers to name variables that are used as counters. For example, at different stages of the program, **`i`** may contain the numbers 1, 2, 3 etc.

- In order to achieve a "counting" effect, you will need to **increment** or **decrement** the value of your counting variable by a set value. Here are some examples:

```
i = i + 1;

i = i - 1;

i = i + 35;

incr = 10
i = i + incr;
```

- To keep things concise, we can use the following shortcuts:

```
i++;  // equivalent to i = i + 1;
i--; // equivalent to i = i + 1;
```

- Counting in JavaScript, like many other programming languages, **often begins at zero**.

- In many cases, this makes a lot of sense, as we will see. However, it can often cause confusion. Consider starting at 0 and counting up to 10. In that case, we may have actually counted 11 items:

  0 (1)
  1 (2)
  2 (3)
  3 (4)
  4 (5)
  5 (6)
  6 (7)
  7 (8)
  8 (9)
  10 (11!)

- If you wanted to give an instruction to someone to perform a repetitive action, you might say that you wanted them to continue the action for a certain number of times. If someone were performing an action 300 times, for example, they might do something like the following to ensure that their count was accurate:

- Write the number 1 on a bit of paper.

- After each action, erase the number on the bit of paper and increment it by 1.

- Before each action, check the number on the bit of paper. If it is less than **or equal to** 300, perform the action.

- Alternatively, they might decide to start counting at 0. In this case, the procedure would be identical, but the check before each action would be to make sure that the number was **strictly less than** 300.

- In JavaScript, we say almost the same thing. The following code will display the numbers 1 to 100 on the page:

```
for     ( i = 1; i <= 100; i++ )
   {
   document.write("<p>" + i " </p>");
   }
```

- The **for** statement tells the browser that we are about to perform a **loop**. The layout here is very similar to a conditional statement, but in this case we have much more information in the brackets. Where our conditional had one JavaScript statement to describe its action, a **for loop** has three:

- An initialiser – this sets up the initial counting condition, in this case `i = 1`.

- A conditional – this is identical to our conditional statements earlier, and must return **true** or **false**. If it returns **true**, the loop continues, otherwise it exits.

- An incrementer – this defines the action to be performed at the end of each loop. In this case, i is incremented by a value of 1.

- The key difference between a conditional and a for loop is that the condition is constantly being changed and re-evaluated. It is possible to create an infinite loop by making the conditional non-reliant on the count value – for example:

  ```
  for     ( i=0; 5 > 4; i++ )
  ```

  will always perform the script in the braces, and will probably cause errors in the browser.

- Note too that it is very common to start counting at zero in JavaScript. The reason for this is that it is often desirable to count **how many times** an operation has been performed. Consider the following:

  ```
  for     ( i=1; 1 < 2; i++ )
  ```

- In this case, the loop will run once, but the value of `i` will be 2, as after the first run, `i` will be incremented to 2, and will then fail the test and so the loop will exit. If we use the following:

  ```
  for     ( i=0; 1 < 1; i++ )
  ```

  Then the loop will run once, and the value of `i` afterwards will be 1, as we might hope.

Project

- Open your previous project file, and save it under the name `chapter_14.html`.

- Clear all JavaScript code from your script tags.

- Write a series of statements to produce a multiplication table as follows:

The 12x Multiplication Table

1 × 12 = 12

2 × 12 = 24

3 × 12 = 36

4 × 12 = 48

5 × 12 = 60

6 × 12 = 72

7 × 12 = 84

8 × 12 = 96

9 × 12 = 108

10 × 12 = 120

11 × 12 = 132

12 × 12 = 144

- The following exercise is more of an HTML example, but demonstrates an important facet of using JavaScript (or, indeed, any programming language) to produce well-formatted text.

- Modify your previous code to make your page's content appear in the centre of the page. Put your multiplication table in an HTML table to make sure that the equals signs, multiplication signs and so forth line up in neat columns:

The 12x Multiplication Table

1 × 12 = 12
2 × 12 = 24
3 × 12 = 36
4 × 12 = 48
5 × 12 = 60
6 × 12 = 72
7 × 12 = 84
8 × 12 = 96
9 × 12 = 108
10 × 12 = 120
11 × 12 = 132
12 × 12 = 144

- As a hint, here is a look at the table cells involved:

The 12x Multiplication Table

1	× 12 =	12
2	× 12 =	24
3	× 12 =	36
4	× 12 =	48
5	× 12 =	60
6	× 12 =	72
7	× 12 =	84
8	× 12 =	96
9	× 12 =	108
10	× 12 =	120
11	× 12 =	132
12	× 12 =	144

15 Arrays

Key points

- In many cases, variables will completely satisfy our data storage needs in JavaScript. However, in a large number of cases, we may wish to "group" variables into a collection of related items.
- Take, for example, days of the week. In each day we perform a number of tasks, so we could want to record each task as a separate item under a group called, say, Monday's Tasks.
- In JavaScript, to achieve this we would store each task in a separate variable, and then group those variables together into an **array**.
- An **array** is a special type of JavaScript object that can store multiple data values – unlike a variable, which can only store one data value at a time.
- It could be helpful to think of an array as a row of mail boxes in an office, just as you might think of a variable as a single, solitary mail box.
- The boxes in an array are numbered upwards, starting at box number 0 – note that counting begins at 0 here, just as we discussed in the previous chapter. The number assigned to each box is known as its **index**.

- In order to use an array in JavaScript, you must first create it. There are a number of ways to create arrays in JavaScript. The simplest follows:

  ```
  arrDays = new Array();
  ```

  This statement creates a new, empty array called arrDays. We can call arrays just like we can variables, but with a few minor adjustments.

- If you already know how many elements a given array will have, you can declare this explicitly:

  ```
  arrDays = new Array(7);
  ```

  This modification creates an array with 7 empty boxes. However, arrays will expand and contract to the required size in JavaScript, so the cases where you will need to state the size of the array are rare.

- More useful, however, is the ability to "fill" the boxes of an array when you create it. For example:

  ```
  arrDays = new Array("Monday","Tuesday");
  ```

  We now have an array with two elements. The first (element 0) has a value of "Monday", while the second (element 1) has a value of "Tuesday". We need not restrict ourselves to string values in arrays – Boolean, numerical and string values are allowed, as in arrays. It is even possible to assign other arrays to array elements – more on this later.

- The most often-used way of creating an array is to use "square bracket" notation. Square brackets play a large role in the use of arrays, so this is often the easiest method to remember:

  ```
  arrDays = ["Monday","Tuesday"];
  ```

  This is identical to the previous example.

- To access an array's elements, we first call the array's name, and then specify the number of the element in square brackets, like so:

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

  Note the lack of quotes around the 0 here. This line of code is equivalent to:

  ```
  alert("Monday");
  ```

  assuming the array is defined as in the previous examples.

- Not only can we access the value of an array element using this notation, but we can also **assign** the value as well:

  ```
  arrDays[2] = "Tuesday";
  arrDays[3] = "Wednesday";
  ```

- If you wish to add an element to an array **without** knowing the index of the last element, you can use the following code:

  ```
  arrDays[] = "some other day";
  ```

- As we will see, arrays are actually just special JavaScript objects, and have properties and methods associated with them. The most important property that every array has is its length property:

  how_many_days = arrDays.length;

- As well as properties, arrays have very useful methods. If you wished to sort your array alphanumerically, you could use the array's sort method thus:

  arrDays.sort();

  Note that the sort method works on the actual array itself, over-writing the current values. So, if you had an array with each day of the week as an element, calling its sort method would mean that `arrDays[0]` was then equal to "Friday", not "Monday".

Project

- Open your previous project file, and save it under the name `chapter_15.html`.

- Clear all JavaScript code from your script tags.
- Write a few JavaScript statements that will present the months of the year on the page in alphabetical order. You should use the following technique to achieve this:
- Store the names of the months of the year in an array.
- Use an array method to sort the array elements alphanumerically.
- Use a **for** loop to *iterate* through each array element in turn, and print the value of the element to the screen (hint, consider the use of **`array[i]`**, where **`i`** is the for loop's counter).
- The above method (the use of a **for** loop to iterate through a series of array elements) is one of the first common programming techniques we have discussed in this course. Its usefulness cannot be overstated, as it allows us to perform repetitive tasks on a series of related elements *without necessarily knowing what those elements might be when we wrote the code*. It can be applied to form elements, cookies, page elements, pages, windows, and just about any other collection of object that you might wish to manipulate with JavaScript.
- To reinforce this generalism, if you have not used the **`array.length`** value in your loop, consider its use now. To prove that you have created a more generic loop, try the code with an array of days instead of an array of months, and see if you have to change any of the looping code.

16 Associative & Objective Arrays

Key Points

- We have already seen that we can access array elements by their index:

```
arrDays = [“Monday”, “Tuesday”];

// print “Monday” to the page
document.write(arrDays[0]);
```

- However, it might be more useful to be able to **name** our array elements. By default, an array will be created as a **numeric** array. We can also create an **associative** array:

```
arrDays = new Array();

arrDays[“Monday”] = “Go to the dentist”;
arrDays[“Tuesday”] = “Attend JavaScript
class”;
arrDays[“Wednesday”] = “JavaScript homework”;

// remind you of Wednesday’s task
alert(arrDays[“Wednesday”]);
```

- This looks a lot like our previous discussion of objects and properties. In fact, since an array *is* actually an object, we can access its elements as though they were properties:

```
// remind you of Wednesday’s task
alert(arrDays.Wedensday);
```

- Note a subtle difference here – in our previous, numeric array examples, the names of the week days were the **values** of our array elements. Here, the names of the week days are the **indexes** of our elements. Avoid the following common error:

```
arrDays = [“Monday”,“Tuesday”];
arrDays[“Monday”] = “Go to work”;

// this is actually equivalent to
arrDays = new Array();

arrDays[0] = “Monday”;
arrDays[1] = “Tuesday”;
arrDays[“Monday”] = “Go to work”;

// and arrDays.length is now 3, not 2
```

Project

- Open your previous project file, and save it under the name **`chapter_16.html`**.
- Clear all JavaScript code from your script tags.
- Write a new script which creates a new, seven element associative array called Week:
- Use the days of the week as indexes for each element of the array.
- Assign a task to each day of the week as each associative element is created.
- Use a for loop to display a calendar on the page, as follows:

  Monday: task
  Tuesday: task

  etc…

- Modify your code to use a prompt box to ask the visitor to choose a day, and display on the page the task allotted to that day.

17 Two Dimensional Arrays

Key Points

- Referring back to our mailbox analogy, where our array could be pictured as a row of mailboxes, each with its own contents and label, a two dimensional array can be thought of as a series of these rows of mailboxes, stacked on top of each other.

- In reality, a **two dimensional array** is simply an array in which each element is itself an array. Each "sub array" of a two dimensional array can be of a different length – in other words, the two dimensional array doesn't have to be "square".

- You can access the contents of each sub array by using two pairs of square brackets instead of just one. An example will illustrate this best:

```
array_1 = ["element", "element 2"];
array_2 = ["another element", 2, 98, true];

array_3 = [array_1, array_2];

alert(array_3[1][3]);  // displays "98"
```

- While you can't mix numerical and string indexing systems in a single array (i.e. an array cannot be both numerical *and* associative), you can have both associative and numerical arrays in two dimensional arrays. For example, consider the above recast as follows:

```
array_3 = new Array();
array_3["firstArray"] = array_1;
array_3["secondArray"] = array_2;

alert(array_3["secondArray"][3]);
   //displays "98" again
```

- Similarly, we can happily use our "objective" notation for associative arrays:

```
alert(array_3.secondArray[3]);
   //displays "98" yet again
```

Project

- Open your previous project file, and save it under the name **`chapter_17.html`**.
- Building on your previous project, create a number of new, seven element associative arrays to represent 4 separate weeks.
- Combine these 4 weeks into a four element array to represent a month.
- Modify your previous code to take a week number and print out all that week's activities to the page.
- Modify one of your week arrays to consist not of single elements, but of arrays of hours from 8am to 5pm. This then represents a *three dimensional array*. We can extend arrays to be ***n-dimensional***, where ***n*** is more or less arbitrary.
- Finally, alter your code to prompt the user for **three** values – a week, a day and an hour. Store these values in three separate variables, and use those variables to display the requested task, or else to display an error message if a task cannot be found.

18 String Manipulation

Key Points

- Throughout your use of JavaScript in a production environment, you will often use it to read values from variables, and alter a behaviour based on what it finds.

- We have already seen some basic string reading in the section on comparisons where we test for equality. However, this all-or-nothing approach is often not subtle enough for our purposes.

- Take the case where we want to check a user's name against a list of known users. If the user enters their name as "Karen", for example, that will be fine if **and only if** they spell the name precisely as we have it recorded, including capitalisation etc. If the user decides to type in her full name, say "Karen Aaronofsky", the code will not recognise her.

- In this case, we want to see if the text "Karen" appears at all in the string. We call this **substring searching**, and it can be incredibly useful.

- One of the simplest substring searches is done by using the **`indexOf`** method. Every string-type variable has this method associated with it. Consider this code:

```
var_1 = "Karen Aaronofsky";
var_2 = var_1.indexOf("Karen");
```

  In this case, the value of **`var_2`** will be 0 – remembering that JavaScript begins counting at 0!

- If we were to search for a surname here:

```
var_1 = "Karen Aaronofsky";
var_2 = var_1.indexOf("Aaronofsky");
```

  var_2 will have a value of 6.

- Finally, if the search were to "fail", so say we searched for the name "Anisa" as a substring, the value of **var_2** would then be -1.

- Note that this is more flexible, but still presents an issue if the user forgets to capitalise any of the substrings that we are searching for – in JavaScript, "Karen" does not equal "karen".

- In order to get around this, we might want to ensure that capitalisation is not taken into account. A simple way to achieve this is to force strings into lowercase before the check is performed:

```
real_name = "Karen";
name = prompt("Your name?","");

real_name = real_name.toLowerCase();
try_name = try_name.toLowerCase();

if ( try_name.indexOf(real_name) > -1 )
{
  alert("Hello Karen!");
}
else
{
 // note we use the original,
 // non-lower-cased name here
 alert("Welcome " + name);
}
```

- There are a number of string methods we can use to perform "value checks" on strings. A few are printed in the following table:

Method	Behaviour
String.indexOf("str")	Returns the numerical position of the first character of the substring "str" in the String
String.charAt(x)	Returns the character at position x in the string – the opposite of indexOf

String.toLowerCase()	Returns a copy of the string with all capital letters replaced by their lowercase counterparts
String.toUpperCase()	Returns a copy of the string with all lowercase letters replaced by their capital counterparts
String.match(/exp/)	Returns true or false based on a **regular expression** search of the string

- The final method here deserves some comment. What is a **regular expression**?
- A **regular expression** is a standard way of writing down a "pattern" of characters that is easily recognisable. For example, consider a typical email address:

  jonathan@relativesanity.com
- An email address follows a "pattern" which makes it instantly recognisable as an email address to a human. Wouldn't it be handy if we could define that pattern in JavaScript for a browser to use? Regular expressions allow us to do just that.
- Let's look at our email address again. We can break it down to a "prototype" email address as follows:

  [some letters]@[some more letters].[a few more letters]
- Of course, it's slightly more complex than that – there are some characters which aren't allowed to be in certain parts of the email address (no spaces, for example), but this lets you see the idea of breaking this string up into required "chunks".
- Now, to convert this to a regular expression. Just as we use quote marks to denote (or "delimit") string values in JavaScript, to signify a regular expression, we use forward slashes: /. Our email address regular expression might look like this:

  ```
  /^.+@.+\..+$/
  ```
- This warrants some discussion. Let's look at this a character at a time:
- / denotes the start of the regular expression

- ^ denotes that we want this regular expression to be found at the very beginning of the string we are searching.
- **.+** the dot symbol is used to stand in for **any** character. The plus signifies we want to find *at least one* of those. So this is equivalent to our plain-English phrase [some letters].
- **@** this is simply a character – it has no special meaning other than to say we want to find an @ character after at least one character from the beginning of the string.
- **.+** the same as before – at least one more character after the @.
- **\.** This is interesting. We know that the dot symbol has a special meaning in a regular expression – it means "match any character". However, here we want to find an **actual dot**. Unlike @, which has no special meaning, we have to tell JavaScript to ignore the dots special meaning. We do this by preceding it with a backslash, which tells JavaScript to treat the character immediately following it as though it has no special meaning. This is a convention you will come across many times while programming. The net result here is that we want to match a dot after a series of characters.
- **.+** and again, at least one more character after the dot.
- **$** this is the mirror of the ^ at the beginning – this matches the end of the tested string.
- / tells JavaScript we are at the end of the regular expression.
- Phew! Lots to consider here. Regular expressions are an arcane art at the best of times, so don't worry too much if the above code is indecipherable. The important thing to realise at the moment is that we can perform some quite sophisticated pattern recognition in JavaScript without having to resort to checking each individual character of a string multiple times.

- The following code checks a variable to see if it looks like an email address:

```
var_1 = prompt("Your email?", "");

if ( var_1.match(/^.+@.+\..+$/) )
{
  alert("valid email address");
}
else
{
  alert("are you sure?");
}
```

- There are a few problems with this code at the moment – for example, it will pass the string "-@-.-" quite happily, which is clearly wrong. We will look at ways around this later on in the course.

Project

- Open your previous project file, and save it under the name **`chapter_18.html`**.
- Clear all JavaScript code from your script tags.
- Use a prompt box to capture some user input to a variable called **`check_string`**.
- Use a document.write statement to output the results of each of the various string methods when applied to the user input.
- Check the user input to see if it's an email address, and alter your output accordingly to either "That's an email address" or "That doesn't look like an email address to me!"
- In the latter case, output the failed string as well so that the user can see their input and modify it next time, if appropriate.

19 Using Functions

Key Points

- A **function** is a named set of JavaScript statements that perform a task and appear inside the standard **<script>** tags. The task can be simple or complex, and the name of the function is up to you, within similar constraints to the naming of variables.

- JavaScript **functions** are declared before they are used. The declaration looks like this:

```
function name_of_function( )
{
  …your code here…
}
```

- Unlike all the JavaScript instructions we have looked at so far, the code inside a function will not be run until specifically requested. Such a request is called a function **call**.

- Functions can be called from anywhere on the page that JavaScript can live, but must be called *after* the function has been declared. For example, if you declare a function in the body of a document and then call it from the head of the document, you may find that the browser returns an error message. For this reason, most JavaScript programmers define any functions they are going to use between **<script>** tags in the head section of their pages to ensure that they are all defined before they are used.

- Functions are to object methods as variables are to object properties – they are also called in a similar manner. To run the code contained in the **name_of_function** function above, we would use the following line of code:

```
name_of_function();
```

- Note the parentheses after the function name. This lets JavaScript know that we are dealing with a function and not a variable. The parentheses also allow us to "pass" extra information to the function, which can alter its behaviour. Consider the following function:

```
function greet_user( username )
{
  message = "Hello " + username;
  alert(message);
}
```

- Whenever the function is called, it will greet the user named. How can we pass this information through? Consider:

```
greet_user("Anisa");
```

or

```
var_1 = prompt("Name?", "");
greet_user(var_1);
```

- We should use functions in our code as often as possible. Whenever we perform an action that isn't accomplished by the use of a method or an assignment operator, we should try to build a function that can accomplish the task.

- For example, we can build a function to check email addresses:

```
function check_email( address )
{
  var_1 = false;
  if ( address.match(/^.+@.+\..+$/) )
  {
    var_1 = true;
  }
}
```

- The above function will take a string that is passed to it (often called the function's **argument**), and will alter the value of **`var_1`** depending on what it finds. However, the function is lacking an important ability – the ability to communicate its findings back out to the rest of the script.

- We have mentioned **return values** a few times in the notes. Now we see a situation that requires a function to **return** its findings to the rest of the code. Ideally, we'd like to be able to use the above function as follows:

```
if ( check_email(address) )
{
  …do some email things…
}
```

- In order for this to work, the return value from check_email would have to be a Boolean value. We can arrange this quite simply:

```
function check_email( address )
{
  var_1 = false;
  if ( address.match(/^.+@.+\..+$/) )
  {
    var_1 = true;
  }
  return var_1;
}
```

- Since var_1 is either true or false, the returned value will be Boolean. We can even skip the use of the variable here and be more direct:

```
function check_email( address )
{
  if ( address.match(/^.+@.+\..+$/) )
  {
    return true;
  }
  return false;
}
```

or even better, since address.match() will return a Boolean value of its own:

```
function check_email( address )
{
  return address.match(/^.+@.+\..+$);
}
```

- The above function may not seem like a great saving. After all, we are using four lines of code to define a function that performs only one line of code. Compare:

```
function check_email( address )
{
  return address.match(/^.+@.+\..+$);
}

if ( check_email(address) )
{
  …do some email things…
}
```

with:

```
if ( address.match(/^.+@.+\..+$/) )
{
  …do some email things…
}
```

- While the benefits here are not obvious, consider the case where, at some point in the future, you discover a better method of checking for email addresses. In the second case abov, you will have to search your code for each and every instance of that method, and replace it with your new method, which may not be one line of code. In the first case, you will just have to change the underlying function definition, and the "upgrade" will be effective throughout your code without you having to update each occurrence.

Project

- Open your previous project file, and save it under the name **`chapter_19.html`**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and one in the body area.
- In the head area, define a function called **`show_message`**. This function should take one argument, called **`message`**, and should use an alert box to display the contents of the argument.
- In the body area, call the function with various messages as arguments.

- Now use a variable in the body area to store the return value of a prompt asking the user for a message. Use this variable as the argument to a single instance of **`show_message`**.

- Define a new function in the head area called **`get_message`**. It should take no argument, but should replicate the function of your prompt in the body area and ask the user for a message via a prompt box.

- Make sure that **`get_message`** **`return`**s a sensible value. We are aiming to replace our prompt statement in the body area with the following code:

  ```
  message = get_message();
  ```

  so consider what you will have to return to enable this to work.

- Once you are happy with your **`get_message`** definition, try replacing your prompt code in the body area with the statement above.

- To demonstrate the power of functions, change the action of **`show_message`** to write the message to the page without changing any code in the body area of the page.

20 Logical Operators

Key Points

- In our discussion of conditionals, we saw how to check the veracity of a single condition via a comparator:

```
if ( x > some_value )
{
  …expressions…
}
```

- We have also seen the limitations of such an approach. Let us say we wanted to discover if **`x`** lay *between* two values, say **`val_1`** and **`val_2`**. There are a number of ways we could achieve this. In our example on student grades, we learned that we could use an **`if…else`** pair to achieve this effect:

```
if ( x > val_1 )
{
  …do something…
}
else
if ( x > val_2 )
{
  …do something else…
}
```

- The above code achieves what we want – for the second branch, x must lie between **`val_2`** and **`val_1`** (assuming **`val_1`** is greater than **`val_2`**, of course). However, it's rather unwieldy, and does not scale elegantly to checking three conditions (say we wanted to check if **`x`** was an even number as well), or in fact to ten conditions.

- Enter **Logical Operators**. These operators are used to join together conditional checks and return true or false depending on whether **all** or **any** of the checks are true.

- In English, we refer to these operators by using the words "AND" and "OR".

- For example, say we wanted to do something each Tuesday at 8pm. We would want to check whether the current day was Tuesday, **and** whether the time was 8pm.

- Another example: Let's say we wanted to do something on the first Tuesday of each month, and also on the 3rd of the month as well. We would have to check whether the current day was the first Tuesday of the month, **or** whether it was the 3rd day of the month.

- Note in the last example, if **both** conditions were true, then we would be on Tuesday the 3rd and would perform the action. In other words, an **or** condition allows for either one, or the other, *or both* conditions to be true.

- In JavaScript, we use the following syntax to check multiple conditions:

  ```
  ( 100 > 10 && 5 < 8 )
  ```

  translates as “if 100 is greater than 10 and 5 is less than 8”. In this case, the result is **true**.

  ```
  ( 100 > 200 && 4 < 9 )
  ```

  in this case, the result is **false**. Note here that only the first condition is actually checked. Since **and** requires both comparisons to be true, as soon as it finds a false one it stops checking. This can be useful.

  ```
  ( 100 > 10 || 9 < 8 )
  ```

  translates as “if 100 is greater than 10 or 9 is less than 8”. In this case, the result is **true**, since at least one of the conditions is met.

  ```
  ( 100 > 200 || 4 > 9 )
  ```

  in this case, the result is **false** since neither of the comparisons are true. Finally:

  ```
  ( 100 > 200 || 5 < 2 || 3 > 2 )
  ```

  in this case, the result is **true**. Any one of the three being true will provide this result.

- As we can see from the last example, this method of checking *scales* to any number of conditions. We can also mix and match the operators. For example:

  ((100 > 200 && 100 > 300) || 100 > 2)

  in this case, the **and** condition evaluates to **false**, but since either that **or** the last condition has to be true to return **true**, the overall condition returns **true** as 100 is indeed greater than 2.

- This sort of complex logic can take a while to comprehend, and will not form a set part of the course. However, it is useful to be aware of it.

Project

- Open your previous project file, and save it under the name **`chapter_20.html`**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and one in the body area.
- Copy the file **`available_plugins.js`** from the network drive (your tutor will demonstrate this), and open it using NotePad's File > Open command.
- Copy and paste the entire contents of **`available_plugins.js`** into your current project file, into the script element in the head area of your page.
- Have a read through the code. Note that it defines a large, two dimensional array. The array has a list of various components that can be present in web browsers (such as Flash or Quicktime)
- Add a function to the head area script element, called **`flash_exists()`**. This function should use a **for loop** to check each of the elements of the **`available_plugins`** array and establish if Flash is present.
- Add a further function to the head area script element, called **`quicktime_exists()`**. This function should also use **a for loop** to check each element of the array, this time returning true if Quicktime is present.
- Finally, add a function to the head area script element called **`both_quicktime_and_flash_exist()`**. This function should call both of the previous functions, store their results in a variable, and produce an alert box containing the message:
- "Both Quicktime and Flash" if both functions returned true; or:
- "One of Quicktime or Flash is missing" if either of the functions return false.
- Call the final function from the body area script element.
- Check your results in your browser.

21 Using Event Handlers

Key Points

- So far, our scripts have run as soon as the browser page has loaded. Even when we have used functions to "package" our code, those functions have run as soon as they have been called in the page, or not at all if no call was made. In other words, the only event our scripts have so far responded to has been the event of our page loading into the browser window.

- Most of the time, however, you will want your code to respond specifically to user activities. You will want to define functions, and have them spring into action only when the user does something. Enter **event handlers**.

- Every time a user interacts with the page, the browser tells JavaScript about an "event" happening. That event could be a mouse click, a mouse pointer moving over or out of a given element (such as an image or a paragraph), a user tabbing to a new part of an HTML form, the user leaving the page, the user submitting a form and so on.

- An **event handler** is a bit of JavaScript code that allows us to capture each event as it happens, and respond to it by running some JavaScript code.

- In general, we attach event handlers to specific HTML tags, so a mouse click on one element of the page might be captured by an event handler, but clicking somewhere else might do something completely different, or indeed nothing at all.

- Some common event handlers are in the table below:

Event Handler	Occurs When…
onload	An element is loaded on the page

onunload	An element is not loaded, for example when a user leaves a page
onmouseover	When the mouse pointer enters the area defined by an HTML element
onmouseout	When the mouse pointer leaves the area defined by an HTML element
onclick	When the left mouse button is clicked within the area defined by an HTML element
onmousedown	When the left mouse button is depressed within the area defined by an HTML element
onmouseup	When the left mouse button is released within the area defined by an HTML element

- The last three are related, but there are subtle differences – onclick is defined as being when **both** mousedown and mouseup events happen in the given element's area. For example, if you click on an area of the page, that registers the area's mousedown event. If you then hold the mouse down and move to another area before releasing, it will register the other area's mouseup event. The browser's click event, however, will remain unregistered.

- In theory, we can add most of these event handlers to just about any HTML tag we want. In practise, many browsers restrict what we can interact with.

- We will mostly be attaching event handlers to **<img>**, **<a>** and **<body>** tags.

- To attach an event handler to a tag, we use the following method:

```
<a href="…" onclick="a_function();">link</a>
```

- We can use this method to attach any event handler listed above to the elements of the page. In addition to calling functions (with any optional arguments, of course), we can write JavaScript directly into our event handlers:

```
<a href="…" onclick="alert('hello');">link</a>
```

- Note the potential issue with quote marks here – if you use double quotes around your event handler, you need to use single quotes within and vice versa.

Project

- Open your previous project file, and save it under the name **chapter_21.html**.

- Clear all JavaScript code from your script tags.

- Ensure that you have a script element in the head area of your document, and none in the body area.

- Within the head area script element, define the following function:

```
function set_status( msg )
{
  window.status = msg;
}
```

- When called, this function will set the text displayed in the browser's status bar (the part of the window below the page content) to whatever is passed as an argument. For example, to set the status bar to display "Welcome", we would call:

```
set_status("Welcome");
```

- Now define the following function immediately below the last:

```
function clear_status( )
{
  set_status("");
}
```

- When called, this function will clear the status bar. Notice that we are using our previous function within the new one. This is a common programming technique that allows us to define functions of specific cases using more general functions.

- Now, add the following HTML to the body area of your page. Remember, we're adding HTML here, not JavaScript, so do not be tempted to use script tags for this part of the project:

```
<a href="#" ↲
   onmouseover="set_status('hello');" ↲
   onmouseout="clear_status();">testing</a>
```

- Load the page in your browser and observe what happens when you move your mouse over the link.

- The # value for the href attribute of the link allows us to define a "dead" link on the page. Clicking on the link will take you nowhere – try it.

- Now, alter the code to have the link point at a real website that you know of.

- Clicking on the link now will take you away from the page. Let's say we want to suppress that behaviour.

- When an event handler intercepts an event, it pauses the normal action of the event. For example, if you used an onclick handler on a link to pop up an alert box, the link would only be followed **after** the alert box had been dismissed. We can use event handlers to cancel the action if required by using their return values.

- Add a new function to the head area script element:

```
function confirm_link( )
{
  check = confirm("This will take you ↵
    away from our site. Are you sure?");
  return check;
}
```

- The value of **`check`** will be **true** or **false**.

- Now, modify your link to contain the following event handler:

```
onclick="return confirm_link();"
```

- By using the word **`return`** in our event handler, the response of the function will be used to decide whether the rest of the normal action is run. In this case, if **`confirm_link()`** returns **false**, our link action will be cancelled.

- Load your page in your browser and view the result.

22 Working with Images

Key Points

- In HTML, we can identify specific elements on the page using an id attribute. For example, to "name" an image, we can use the following code:

  ```
  <img src="logo.gif" alt="" id="theLogo" />
  ```

- To refer to this element in JavaScript, we can now get to it directly by its id value:

  ```
  document.getElementById("theLogo")
  ```

- This method will return an object that refers to the given element on the page. If no such element can be found, the method will return false.

- For easier use, we can assign the object found to a variable. For example, to create an object called our_logo in our scripts, we can use the following line of code:

  ```
  our_logo = document.getElementById("theLogo");
  ```

- The resultant object has a number of properties. Since, in this case, our object represents an image element, its properties include:

  ```
  our_logo.height
  our_logo.width
  our_logo.src
  ```

- We can use JavaScript to change any of these properties, so if we wanted to change the image displayed, we could do so as follows:

  ```
  our_logo.src = "new_logo.gif";
  ```

Project

- Copy the folder called **images** from the network drive to your project folder.
- Open your previous project file, and save it under the name **chapter_22.html**.
- Clear all JavaScript code from your script tags.
- Ensure that you have a script element in the head area of your document, and none in the body area.
- Within the body area of your page, create an image element that loads an image from the **images** folder. Give the element an appropriate **id** attribute.
- In the head area script element, define a function called describe_image() that will pop up an alert box containing the following information about your image:

  the image file used
  the image width
  the image height

- To have each bit of text appear on a separate line, you can add the following character to your alert text:

  ```
  \n
  ```

  for example

  ```
  alert("line one\nLine two");
  ```

- Load your page in the browser and view the results.

23 Simple Image Rollovers

Key Points

- A simple image rollover is an effect which happens when the mouse appears over an image (usually a linked button) on the page. The original image is replaced by another of equal size, usually giving the effect that the button is highlighted in some way.

- With what we have learned so far, we already have the ability to create a simple image rollover effect. All that remains is to clarify the particulars of the process:

- The page is loaded and the original image appears on the page as specified by the **`<img>`** tag's **`src`** attribute.

- The mouse moves offer the image and the alternative image is loaded into place.

- The mouse leaves the image and the original image is loaded back.

- As you may have realised, we are going to use JavaScript to alter the **`src`** attribute of the image tag. The best way to think of this is to picture the **`<img>`** tag as simply a space on the page into which an image file can be loaded. The **`src`** attribute tells the browser *which* image to load into the space, and so if we change that value, the image will be changed.

- In other words, the **`id`** attribute of the **`<img>`** tag is naming the "space", not the image.

- Now, in order to alter the src attribute with JavaScript, we need to tell JavaScript which image "space" we want to alter. We use the id attribute along with the getElementById() method from the last chapter to do this:

```
button_img = ↲
    document.getElementById("button");

button_img.src = "new_image.jpg";
```

- We can directly insert this code into the image's event handler:

```
<img src="old.jpg" id="button" ↲
    onmouseover=↲
    "document.getElementById('button').src ↲
     = 'new.jpg';">
```

- Note that this code is suddenly very convoluted. There are two immediate potential solutions. The first is to define a function:

```
function swap_image( id, new_image )
{
  img = document.getElementById(id);
  img.src = new_image;
}

…

<img src="old.jpg" id="button" ↲
    onmouseover= ↲
    "swap_image('button', 'new.jpg');">
```

- This is a much cleaner solution, and more importantly we can use this for any images on the page, simply by changing the arguments of our function call. We can also use the function to achieve the "swap back" functionality:

```
…

<img src="old.jpg" id="button" ↲
    onmouseover= ↲
    "swap_image('button', 'new.jpg');" ↲
    onmouseout= ↲
    "swap_image('button', 'old.jpg');">
```

- We can go even further in "cleaning up" our code, though. Because the event handler is being used to alter the object which is experiencing the event (ie, the mouse is moving over the image tag that we are trying to change), we can use the "automagic" JavaScript object **this** to perform the operation:

```
function swap_image( img, new_image )
{
  img.src = new_image;
}

…

<img src="old.jpg" id="button" ↵
    onmouseover= ↵
    "swap_image(this, 'new.jpg');">
```

- Note a couple of things. Firstly, **this** has no quotes around it – we are using it like a variable name. Secondly, our function now uses the first argument directly, instead of using it to get the relevant object from the page. We can do that because **this** is actually an object – it's an object that takes on the properties and methods of whatever object it is called from, in this case, it becomes equivalent to **document.getElementById('button')**, although is obviously much shorter!

- Using **this** has some limitations. For example, if we wanted to change the **src** attribute of any other image on the page when the mouse moved over "**button**", we would be unable to use this, and hence would have to define another function that could take an id as its argument and get the relevant object that way.

Project

- Copy the folder called **buttons** from the network drive to your project folder.

- Open your previous project file, and save it under the name **chapter_23.html**.

- Clear all JavaScript code from your script tags.

- Ensure that you have a script element in the head area of your document, and none in the body area.

- Within the body area of your page, create a paragraph containing six image elements. Set the **src** attributes of the images to load the following files in your copied **buttons** folder, and give each a sensible **id**:
- **contacts.jpg**
- **home.jpg**
- **people.jpg**
- **products.jpg**
- **quotes.jpg**
- **whatsnew.jpg**
- Create a JavaScript function in the head area script element that takes two arguments – an **id** and a file name. It should alter the src property of the appropriate image object to the file name given.
- Use this function to swap the **src** attribute of the contacts button to **contactsover.jpg** when the mouse moves over the image.
- Once you have this working, update the remaining five images with event handlers to swap their **src** attributes to their appropriate "over" image.
- Add event handlers to all six images to ensure that they return to their original state when the mouse is moved away from them. Check your work in your browser
- Add a new paragraph above the previous one, and add an **<img>** tag to it to containing the file **rocollogo.jpg** from the images folder.
- Add a text link to a new paragraph between the two paragraphs. The link should be a "dummy" link (ie use "**#**" as its **href** attribute value), but when the mouse moves over it, the image above it should change to show **rocollogo.gif**.
- Moving the mouse away from the text link should return the logo to its previous state.
- Check your work in your browser.

24 Object Instantiation and Better Rollovers

Key Points

- So far we have seen a very simple example of an image rollover. It is functional and works as desired, but it is lacking in a few finer details.
- Specifically, when we use JavaScript to change the src attribute of an <img> tag, the browser has to load this image from scratch. On our local machine, this will not cause an appreciable delay, but when dealing with a remote server (as we will be on the Internet), this delay can lead to a noticeable "lag", which can destroy the feeling of a dynamic interface.
- Ideally, we would like to instruct the browser to load any alternate images when it loads the page. This will allow us to ensure that the new images are sitting on the user's computer, ready to be swapped in and out instantly.
- To do this, we need to look at **object variables** and **object instantiation** (or *creation*). In particular, we need to look at the Image object.
- Each **`<img>`** tag on the page has a corresponding Image object in JavaScript. We have looked at ways of manipulating this directly, and have an idea of some of the properties an Image object can have.
- However, we can create Image objects directly in JavaScript, without the need for a corresponding **`<img>`** tag on the page. When we create such an object, and set its src property, the browser will load the appropriate file into memory, but will not display the image on the page.

- In other words, we can create "virtual" images that exist within JavaScript, and use these Image objects to store the alternate images for our "real" images.

- To create an Image object (in fact, to create *any* object), we need to use the following code:

```
virtual_image = new Image();
```

- We have seen this sort of syntax before – the use of the **new** keyword when we created our Arrays. **new** tells JavaScript that we are creating a new object. The **virtual_image** part of the assignment is just a variable name. In this case, the variable is an **Object Variable**, since it contains an object.

- To use this variable to preload our images, we take advantage of the fact that it has the same properties and methods as any other image:

```
virtual_image.src = "contactsover.jpg";
```

- The browser will now preload our image, ready to be swapped in at a later time.

Project

- Open your previous project file, and save it under the name **chapter_24.html**.

- Starting with your previous code, create a new function called preload_images in the head area script element of your page.

- Use this function to create seven new image objects, and use each objects corresponding object variable to preload your "over" image variations.

- Check your work in your browser to ensure that the image swapping still works as expected.

- Add your preload_images function to the body tag of your page to ensure that it runs when the page has finished loading. Use the following syntax:

```
<body onload="preload_images();">
```

- Once you have verified that the image swapping still works as expected, expand your preload_images function to define an array of images to preload, and then use a for loop to move through the array and assign each image to the src property of an object variable. ***Hint***: an object variable can be anything that can store information – for example an array element.

- Check your work in your browser.

25 Working with Browser Windows

Key Points

- Using standard HTML links, we can open new browser windows:

  ```
  <a href="#" target="_new">link</a>
  ```

- The amount of control this method affords us over the resultant image, however, is nil. We cannot control the size, shape, location on the screen or anything else about that window with this method.

- JavaScript allows us much finer control, as we may expect:

  ```
  window.open( page_url, name, parameters );
  ```

- As we can see from the above prototype, there are only three arguments that this method can take. However, the **`parameters`** argument is actually more complex than we might assume:

  ```
  "param1,param2,param3,param4…"
  ```

  since we can use it to add many parameters to the method. Note there are no spaces in the parameter list.

- **`name`** is used to give the window an HTML name – so we can use that name to open links into the new window using the "**`target`**" method above.

- The return value from the **open** method is an object variable referring to the newly opened window. In other words, if we open a new window like so:

```
win = window.open("page.html", "", "");
```

  we can use the object variable **win** to alter the new window through JavaScript.

- A practical example – let's say we want to open a new window 300 pixels high by 400 pixels wide, and display the BBC news page in it. The following code would suffice:

```
window.open("http://news.bbc.co.uk/", ↲
    "bbc", "width=300,height=300");
```

- Some available parameters are given in the table below:

Parameter	Value	Function
location	Yes/no	Specifies whether or not the location (address) bar is displayed
menubar	Yes/no	Specifies whether the menu bar is displayed
status	Yes/no	Specifies whether the status bar is displayed
width	Pixels	Specifies the width of the new window
height	Pixels	Specifies the height of the new window
resizable	Yes/no	Allow or disallow window resizing
scrollbars	Yes/no	Allow or disallow window scrolling

- If no parameters are set, then the value of each parameter is set to "yes" where appropriate. For example:

```
window.open("http://www.bbc.co.uk/", ↲
    "bbc", "");
```

  is equivalent to:

```
<a href="http://www.bbc.co.uk/" target="bbc">
```

- However, if **any** parameter is set, all others default to "no". So if you wanted to have a scrollbar but nothing else, the following would suffice:

```
window.open("http://www.bbc.co.uk/", ↲
    "bbc", "scrollbars=yes");
```

Project

- Open your previous project file, and save it under the name **chapter_25.html**.
- Remove all functions and HTML from your previous file, leaving only the logo image and the link.
- Create a function in the head area script element called **view_new_logo**. This function should:
  - Open a new window 200 pixels square.
  - Load the image **rocollogo.jpg** from the images folder.
  - Be called **RocolLogo**
  - Be stored in an object variable called **objNewLogo**.
  - Have no scrollbars, be of a fixed size, and have no other features where possible.
- Remove all event handlers from the link, and add a new one to run the function above when the link is clicked.
- Once you have verified that a window pops up as required when the link is clicked, test each parameter from the table above in the function.

26 Positioning Browser Windows

Key Points

- The **screen** object provides you with access to properties of the user's computer display screen within your JavaScript applications.

- Some of the available properties are:

Property	Description
availHeight	The pixel height of the user's screen minus the toolbar and any other permanent objects (eg the Windows Taskbar)
availWidth	As availHeight, but dealing with horizontal space
colorDepth	The maximum number of colours the user's screen can display (in bit format, eg 24 bit, 16 bit etc)
height	The true pixel height of the user's display
width	The true pixel width of the user's display

- The **`left`** and **`top`** parameters of the **`open()`** method enable you to specify the position of a window on screen by specifying the number of pixels from the left and top of the screen respectively.

- If you need to use a variable to specify the value of a parameter in the **`open()`** method, you would do so as follows:

```
window.open("index.html", "window_name", ↲
    "width=200,height=200,left="+var_left+ ↲
    "top="+var_top);
```

Where **`var_left`** and **`var_top`** are the appropriate variables.

- Note the use of the **+** operator to ensure that the third parameter in the **`open()`** method remains as one string when variables are added.

- In order to centre the window on the user's screen, a little simple geometry is required. The centre of the screen is obviously the point found when we take the width of the screen divided by two, and take the height of the screen divided by two. However, if we set the window's top and left values to these coordinates, the top left **corner** of the window will be centred, not the window itself.

- In order to centre the window, we need to subtract half of the window's height from our top value, and half of the window's width from our left value. A simple script to accomplish this is as follows:

```
win_width = 200;
win_height = 200;

win_left = (screen.availWidth/2) ↲
    - (win_width/2);
win_top = (screen.availHeight/2) ↲
    - (win_height/2);
```

- By using this script, the values of win_left and win_top will be set correctly for any window using win_width and win_height appropriately to be centred on the screen.

Project

- Open your previous project file, and save it under the name **`chapter_26.html`**.

- Modify your existing code to ensure that the logo appears centred on the user's screen. If possible, do not modify your original function by doing anything more than two new functions – **`get_win_left( width )`** and **`get_win_top( height )`**.

27 Focus and Blur

Key Points

- The **`focus()`** method of the **`window`** object gives focus to a particular window. In other words, it ensures that the window is placed on top of any other windows, and is made active by the computer. For example, if we have created a new window and stored the result in an object variable called **`new_window`**, we could ensure that the window was brought back to the front at any point after it had been opened by using the following code:

```
new_window.focus();
```

- Conversely, we can use the **`blur()`** method to remove focus from the specified window – returning focus to the previously selected one as appropriate. Its use is similar to the **`focus()`** method.

- Both these events have associated event handlers: **`onblur`** and **`onfocus`**. However, since they do not have associated HTML tags, how can we attach event handlers to the object?

- It turns out that, within JavaScript, each object has an individual property for each event handler it can have applied to it. For example, if we wanted to add an event handler to an image tag on the page, we could apply either of the following methods to do that:

```
<img src="…" id="test_img" ↵
onclick="do_something();" />

<script>
document.getElementById("test_img").onclick=↵
    do_something();
</script>
```

- So, to attach a function to a window's **`focus`** event, we could use:

```
new_window.onfocus = some_function();
```

Project

- Open your previous project file, and save it under the name **`chapter_27.html`**.
- Modify your function to open another window as well as the original one, with the following features:
  - The second window should be called **`oldRocolLogo`**, should be assigned to the object variable **`objOldLogo`**, and display the old logo from the file **`images/rocollogo.gif`**.
  - When opened, the windows should be positioned so that both can be clearly seen.
- Test your modifications at this point.
- Observe the function's action when the windows are opened, then the original window is placed in front of them and the function is invoked again.
- Add a new statement to the function which uses the **`focus()`** method to ensure that when the function is called, both new windows are moved to the top of the "stack" of windows.
- Observe which logo appears "on top" when the function is called. Use the **`focus()`** method again to alter this.

28 Dynamically Created Content

Key Points

- It's quite easy to create a new page on demand using JavaScript. In this context, we are talking about creating a completely new page in a window without loading any file into that window.

- To do this, invoke the **`open()`** method of the **`window`** object, leaving the location parameter empty:

```
new_win = window.open("", "newWin", ↲
    "params…");
```

- Next, remember that you can write HTML code to a page using the window's **`document.write()`** method. Up until now, we have used only the current window's **`document`** object. However, we can specify *which* window's **`document`** we want to manipulate as follows:

```
document.write();  // write to current window
new_win.document.write(); // or new window
```

  For example:

```
new_win.document.write("<html><head>");
new_win.document.write("<title>demo</title>"):
new_win.document.write("</head><body>");
new_win.document.write("<h1>Hello!</h1>");
new_win.document.write("</body></html>");
```

Project

- Open your previous project file, and save it under the name **`chapter_28.html`**.

- Modify your existing script to create a third window with the following properties:
  - The window should be called **`newHTML`**, and be assigned to the object variable **`objNewHTML`**.
  - It should be 400 pixels square.
  - It should not load any page when it is created. It should display the word "Welcome" as an **`H1`** header.
  - It should contain a paragraph with the text "Please decide which logo you would like to choose."
  - The third window should carry the title "Rocol Art"
  - When all windows have been opened, the third window should be **`focus`**sed.
- Check your work in the browser.

29 Working with Multiple Windows

Key Points

- The window object's **`close()`** method enables you to close a window. If you have an object variable in the current window referring to the window you wish to close, you can simply use:

  ```
  new_window.close();
  ```

  to close the window.

- Things are a little more complicated when you wish to close a window from a window other than the one which opened the new window. In order to tackle this, we need to think about **window scope**.

- When we write JavaScript into our code, all functions and variables within that script are available for us to use **within that window**. For example, if we have a function called **`say_hello()`** in our main window, we can easily call that function. However, if we want to call the function from a newly opened window, we cannot call it directly from the new window, as our functions and variables are "tied" to the windows in which they were first defined.

- This is why, when we want to write to any window other than the one containing our JavaScript code, we must access the **`document`** object *of that window* in order to put content in the right place,

- But how about in the other direction? Let's say we use a function in our main window (call it **window 1**) to open a new window (**window 2**). If we store **window 2** as an object variable, we can access all properties of **window 2** from **window 1** by using that object. The question is, how do we access any properties of **window 1** from **window 2**?

- The key is that **window 2** and **window 1** have a special relationship – a "parent/child" relationship. We can access any property of **window 1** from **window 2** by using the special object called **`opener`**. This is an object created within any window that has been opened by JavaScript, and it always refers to the window that opened the new window.

- To illustrate this, let's consider our previous project. We have three new windows, and one "parent" window. If we wanted to use an event handler on one of the new windows to close on of the other new windows, we would need to first access the parent window, and then access the object variable within the parent window that pointed to the window we wanted to close.

- Let's say we wanted to close window_2 from a link in window_1. We would have to create an event handler in the link with the following code:

```
onclick="opener.window_2.close();"
```

Project

- Open your previous project file, and save it under the name **`chapter_29.html`**.

- Modify your existing script to achieve the following:

  - Add two new paragraphs to the third window's content containing the following text:

    The Old Logo

    The New Logo

  - Each line should be contained in a hyperlink whose event handler accesses the parent window's object variable pointing to the appropriate new window. Its **`close()`** method should then be invoked.

- Check your work in the browser.

30 Using an External Script File

Key Points

- JavaScript can easily save us from having to type lots of HTML. As we have seen, we can use JavaScript to generate large form elements and tables using a small amount of code. This is good news for us, as it makes our work easier. It is also good news for our visitors, as they only have to download a small amount of code for a relatively large amount of content.

- However, we can do better. As it is, if we have one function that will generate a year drop down menu for a form's date of birth section, we need to include that function in every page that requires it. This means that our visitors are downloading identical content multiple times. In addition, if we change or improve our function, we have to ensure that we update that function in every page that has it included.

- HTML allows us to solve this problem by providing a mechanism to load an external text file into the HTML page and treat its contents as JavaScript code. Since it is a separate file, once it has been downloaded once, the browser will not download further copies of the file if it is requested by another page. In other words, we can load all our JavaScript code into one file, and any changes there will instantly be reflected across the entire site.

- To use code in an external file, we still use the **<script>** tag – but with a new attribute, the **src** attribute. This is very similar to loading an image file on to a page:

```
<script language="JavaScript" ↲
    type="text/JavaScript" ↲
    src="s/script_file.js"></script>
```

- Note three things:

- The language and type attributes are essential here.
- The script tag still has a closing **`</script>`** tag.
- We cannot add any further JavaScript between the tags when we are using the tags to load an external file. To add JavaScript to the current page only, we have to use a second set of **`<script>`** tags.

Project

- Open your previous project file, and save it under the name **`chapter_30.html`**.
- Move all your JavaScript function definitions, and any other code in the head section script element to a new file called **`script.js`**.
- Modify your head section script element to load code from the new file.
- Check that your page still works as expected.
- **NOTE**: the **`.js`** file extension is just a naming convention. **`<script>`** tags will load JavaScript from any text file (hence the need to include the type and language attributes). However, it is a widely used convention, and it is worth sticking to in order to keep your code easily understood by anyone who may work on your code in the future.

31 Javascript and Forms

Key Points

- Without JavaScript, the server handles all validating and processing of information submitted via a form. Using JavaScript on the client side of the equation saves time for the user and creates a more efficient process.

- Some processing can be handled by JavaScript (for example, mathematical computations) and JavaScript can ensure that only correct data is sent to the server for processing.

- JavaScript is used in conjunction with server-side processing – it is not a substitute for it.

- To access information in a form we use the document object's **`getElementById()`** method, as we have previously to access other objects on the page. For example, if we have a form on the page like so:

  ```
  <form id="testForm" … >

  </form>
  ```

  we would access it in JavaScript by using:

  ```
  document.getElementById('testForm');
  ```

- The resultant object is actually a multi-dimensional array. Each of its elements is itself an array containing information about the elements of the form (text boxes, buttons etc). By properly naming each of the form's elements in the appropriate <INPUT> tags, you can access information relating to each of the form's elements.

- To access the data stored in a text box called **Name** which is included in the form with the id **Enquiry** you use the value property like so:

```
objForm = document.getElementById('Enquiry');

strValue = objForm.Name.value
```

- There are two ways of sending form data to the server. Using the **method** attribute, you can specify either the GET or the POST methods:

```
<form id ="enquiryform" method="GET"…
```

or

```
<form id ="enquiryform" method="POST"…
```

- In general, you should use the POST method if you want to send a lot of data (eg files, large amounts of text) from your form. You should use GET if you want to process search forms etc, as a GET form will be transmitted just like a URL, and is hence "savable" as a bookmark or link.

- In general, you will send your form to a server side script, specified by the form's action attribute:

```
<form id="enquiryform" method="GET"
→ action="process.php"…
```

- When the user clicks on a form's Submit button, without JavaScript intervention, the form's data is sent straight to the server for processing. But you can intercept the data (so you can process it with JavaScript) before it is sent, by including the **onsubmit** event handler in the **<form>** tag. This enables you to run a JavaScript function before the data is sent:

```
<form id="enquiryform" method="GET"
→ action="process.php"
→ onsubmit="functionName()">
```

- In the above example, when the user clicks on the Submit button, the function **functionName()** is run first, then the data is sent to the server.

- When the submit event is triggered by the form's submission, the browser waits to discover what is *returned* from the event handler. By default, the event handler will return **`true`**. However, if the event handler returns a **`false`** value, the form's submission will be aborted, and the page will not be submitted to the server.

- By returning a value of either true or false from your function (**`functionName()`** in the above example), and ensuring that this is also the return value of the **`onsubmit`** event handler, you can decide whether or not the form's data is actually sent to the server.

- You specify that the return value of the function is also the return value of the **`onsubmit`** event handler in the following way:

```
onsubmit="return functionName();"
```

Project

- Open your previous project file, and save it under the name **`chapter_31.html`**.

- Clear any content from the body element of the page, and ensure that the head area script element has no code in it.

- Save a copy of this page as **`processing_31.html`**, and put an **`<h1>`** element in the body area saying "**success**!".

- Now, close your **`processing_31.html`** page and create a form on your original **`chapter_31.html`** page using HTML – if you have difficulty with this, the tutor will provide an example to duplicate. Your form should:

  - have an id of **`jsCourseForm`**.

  - have a single input box with an name value of **`name`**.

  - have a submit button.

  - use the GET method of submission.

  - have **`processing_31.html`** as its **`action`** attribute.

- have an onsubmit event handler that returns the value of the (as yet non-existent) **`check_form()`** function.

- Now, create a function in the head area script element called check_form(). This function should:
  - use the document's getElementById() method to store a reference to the form's object in an object variable.
  - store the value of the form's **`name`** element in another variable.
  - if the value of the name element is not "**Bugs Bunny**", an alert box should appear stating:

    ```
    Sorry chum!
    Either you misspelled your name…
    Or you haven't got what it takes…
    Try again.
    ```

  - in addition, the form should be prevented from submitting.
- If the user enters the correct name, however, the form should submit without interruption.
- Check your work in your browser.

32 Form Methods and Event Handlers

Key Points

- Each form object (e.g. text, button etc) has a set of properties associated with it. This is different for each form element. The **`value`** property is common to most form elements and is one of the most useful properties.

- You can assign the data stored in a text box called **`Name`** which is included in the form with id **`Enquiry`**, to a **`variable`** like so:

```
variable = document.getElementById('Enquiry').
→ Name.value
```

- Form objects also have methods associated with them. The set of available methods is different for each form object.

- Below is a list of commonly used methods for the text object:

Method	Description
blur()	Removes the focus from the text box
focus()	Gives the focus to the text box
select()	Selects the text box

- Below is a list of commonly used methods for the button object:

Method	Description
blur()	Removes the focus from the button
focus()	Gives the focus to the button
click()	Call's the button's onclick event handler

- Form objects also have event handlers associated with them. The set of available event handlers is different for each form object.

- Below is a list of commonly used event handlers for the text object:

Event handler	Runs JavaScript code when…
onblur	The text box loses the focus.
onfocus	The text box receives the focus.
onselect	The user selects some of the text within the text box..
onchange	The text box loses the focus and has had its text modified.

- Below is a list of commonly used event handlers for the button object:

Event handler	Runs JavaScript code when…
onBlur	The button loses the focus
onFocus	The button receives the focus
onClick	The user clicks the button

- Finally if you are sending data to the server, a submit button is not the only way. You could use the submit() method in a function which is invoked by an event handler. This operates as if the Submit button was clicked:

```
document.getElementById('Enquiry').submit();
```

Project

- Open your previous project file, and save it under the name **`chapter_32.html`**.

- Modify your form in the following way:

  - Remove the submit button

  - Replace the submit button with a standard form button.

  - Add an event handler to this button to invoke the function in the head area script element.

  - Remove the **`onsubmit`** event handler from the form element.

- Now, modify the **`check_form()`** function in the following way:
  - If the user types in the name “Bugs Bunny”, the function submits the form using the form’s submit() method.
  - If the user types anything else, the previous alert box warning is displayed and the form is not submitted, but also:
    - The words “please try again” are displayed in the text box.
    - The text box is given focus.
    - The text in the text box is selected.
- Check your work in your browser.

33 JavaScript and Maths

Key Points

- The **`Math`** object is a pre-defined JavaScript object containing properties and methods which you can use for mathematical computation.

- Below is a selection of some useful **`Math`** methods:

Method	Returns
Math.cell()	The smallest integer greater than or equal to a number. That is, it rounds up any number to the next integer. **`Math.cell(2.6)`** returns 3 and so does **`Math.cell(2.2).`**
Math.floor()	The largest integer greater than or equal to a number. That is, it rounds down any number to the next integer. **`Math.floor(2.2)`** returns 2 and so does **`Math.floor(2.6).`**
Math.max(n1,n2)	The larger of the two arguments.
Math.min(n1,n2)	The smaller of the two arguments.
Math.random()	A random number between 0 and 1.
Math.round()	The number rounded to its nearest integer.
Math.sqrt()	The square root of a number.

- You don't need to include a Submit button in a form and you don't need to send form data to the server. You could use event handlers to invoke JavaScript code which merely processes the data on the form (e.g. you may just perform some mathematical computations on some user data).

- If you are not sending the data to the server, there is no need to include either the Action or Method attributes in the <FORM> tag, though by default the Action will usually submit to the current page, and the Method will be set to "get".

Project

- Open your previous project file, and save it under the name **`chapter_33.html`**.
- Remove all content from the body section of the page.
- Copy the file **`max_wins.html`** from the network, and open it using NotePad's File > Open command.
- Copy and paste the entire contents of **`max_wins.html`** into your current project file, into the body element of the page.
- Remove all JavaScript code from the script element in the head section of the page.
- Take some time to open your project file in a browser, and study the code. This project will enable the page to:
  - Generate two random numbers when the button is pressed.
  - Display the random numbers in the text boxes marked **player 1** and **player 2**.
  - Compare the two random numbers and display the name of the player whose number is higher in the text box marked **winner**.
- When studying the code, note that the form has no valid action or method attributes. It also has no "submit" button. We are not going to allow the form to submit to the server at all, but are going to use JavaScript to do all our processing on the form.
- Now, modify the HTML code on the page to add an event handler to invoke a new function defined in the head area script element. The function should perform the following operations when the form button is clicked:
  - Place two separate random integers between 0 and 100 in each of the Player text boxes.

- Find a way to use a Math method to compare the two entered integers. Once compared, the function should then place the appropriate value of **Player 1**, **Player 2**, or **Draw** in the Winner text box.

- Check your work in your browser.

34 Object Variables – A Refresher

Key Points

- Referring to objects can be a lengthy process. Consider the Player 1 text box in the previous example. You refer to it in full as follows:

```
document.getElementById("MaxWins").Player1
```

- This notation although precise, is tedious, time consuming and can be prone to error. Luckily, there are some shortcuts which can save time and reduce typing errors.

- We can use an object variable to simplify our work whenever we are in situations where certain objects need to be referred to repetitively. To use an object variable, you begin by simply assigning it a specific object:

```
oPlayer1 =
→ document.getElementById("MaxWins").Player1
```

- Once assigned, you can use the object variable in any situation where you would use the specific object itself. Using the object variables from the previous paragraph:

```
oPlayer1.value
```

is the same as:

```
document.getElementById("MaxWins").Player1.value
```

- Bear in mind that you assign objects to object variables. You would assign a text box or a button or an image etc to an object variable and then refer to that object's properties as shown above **`(oPlayer1.value).`** You don't assign text or string values to object variables. So:

```
oPlayer1 =
→ document.getElementById("MaxWins").Player1.value
```

  would merely assign the *value* stored in **`Player1`** to the variable **`oPlayer1`**.

- Hopefully it is obvious that you don't need to include the 'o' at the beginning of the object variable's name. It's just a convention to help distinguish between the different types of variables.

Project

- Open your previous project file, and save it under the name **`chapter_34.html`**.

- Modify the function in the head area script element in the following way:

  - All specific object which are referred to more than once are each assigned their own object variable at the start of the function.

  - References to specific objects in the code are replaced by the appropriate object variables.

- Test your work in your browser to ensure that it functions as before.

35 Actions From Menu Items

Key Points

- The HTML **<select>** form tag enables you to create a menu (select box) of options from which the user can choose:

```
<form id="menu">

<select name="Product">
  <option value="one">Image one</option>
  <option value="two">Image two</option>
  <option Value="three">Image three</option>
</select>

</form>
```

- Ordinarily, selecting an option from a select box merely specifies the value of the select box. This is then used for further processing – either by JavaScript or by the server.

- You can use JavaScript to invoke actions based on the current value of a select box (the selected option). These actions might be directly loading another page or performing some other type of processing.

- Another name for this type of action is a "jump menu".

- In order to do this, you need to know that each select box on a form has a select object associated with it. Using the above example, you can therefore access the current value of the Product select box (ie the value of its currently selected option) which is located on the ProductMenu form, in the following way:

```
document.getElementById("menu").Product.value
```

- As you would expect, the select option has methods and event handlers associated with it:
  - The focus() method gives the focus to a select box.
  - The blur() method removes the focus from the select box.
- The available event handlers for the select object are given below:

Event handler	Runs JavaScript code when….
onblur	The select box loses the focus
onfocus	The select box receives the focus
onchange	The select box has had its value modified

- The way to invoke action(s) when the user selects an option from a select box is to prepare a JavaScript function which will be invoked using the select object's **onchange** event handler:

```
<select name="name" onchange="functionname()">
```

Project

- Open your previous project file, and save it under the name **chapter_35.html**.
- Clear the head section script element of JavaScript, and remove all content from the body area of the page.
- Create a form in the body area of the page. Give the form an id of **jumpMenu**.
- In the form, place a select element. Give the select element the following values and labels eg:

```
<select name="menu">
   <option value="value">label</option>
   …
</select>
```

Value	Label
http://www.bbc.co.uk/	The BBC
http://www.google.com/	Google
http://www.hotmail.com/	Hotmail

http://www.ed.ac.uk/	The University of Edinburgh
http://www.apple.com/	Apple Computer
http://www.microsoft.com/	Microsoft Corporation

- Finally, add an option element to the beginning of your menu like so:

  ```
  <option value="">Select a destination</option>
  ```

- We are now going to use the menu's **`onchange`** event handler to invoke a function we are about to define. The function is to be called **`jump_to_page()`**.

- Define such a function in the head area script element. The function should:

  - Create an object variable referencing the form element that represents the menu.

  - Get the value of the menu at that moment.

  - If the value is not equal to "" (ie, if a valid selection has been made), the function should use the use JavaScript to load the selected page into the browser.

- Check your work in your browser.

36 Requiring Form Values or Selections

Key Points

- Form data may be invalid if it is sent to the server without certain information – for example, if the user has omitted to select an item from a menu. Better not to bother processing, than to waste the time of the user and server by trying to process the invalid information.

- In the case of a selection box, one way of validating is to include a null value for the default option. In the previous project, the default value of the selection box (always the first option unless specified otherwise) was "". If the user doesn't actively make another selection, then you can check the value of the selection box before sending it to the server.

- Another example is the case of a set of radio buttons. Imagine that for an imaginery company, an order form contained two form elements – a selection box to specify which product was being ordered and two radio buttons to specify whether it was being ordered as a photographic print or as a slide.

- Let's say the id of the order form is **`OrderForm`** and the name of each radio button is **`Format`**. (Remember that in HTML, radio buttons in a set are related to each other by their name that must be the same for each related radio button).

- The radio object is an array where each element of the array stores information relating to each of the radio button objects (remember, counting starts at 0).

- As the name is identical for each radio button in the set, you can't access an individual radio button by using its name. But you can access it using the standard array notation:

  ```
  oOrderForm.Format[i]
  ```

  where in this case, I is an integer between 0 and 1, and **oOrderForm** is an object variable pointing to the form element.

- Radio buttons in a set each have a checked property that stores a Boolean value specifying whether or not a radio button is checked. Obviously, you can access this value in the following way:

  ```
  oOrderForm.Format[i].checked
  ```

- So, to verify that in a set of radio buttons, at least one of them is checked, all you have to do is loop through each of the radio buttons using the array's length property to specify the number of iterations of the loop:

```
FormatSelected = false;
oOrderForm =
→ document.getElementById("OrderForm");

for ( i=0; i < oOrderForm.Format.length; i++ )
{
    if ( oOrderForm.Format[i].checked )
    {
        FormatSelected = true;
    }
}
```

- In our order form example, one of the products might only be available as a slide. You could use the **onchange** event handler of the **select** object to invoke a function which automatically sets the value of the relevant radio object's checked property:

```
function SetFormat ()
{
    oOrderForm =
    → document.getElementById("OrderForm");
    if ( oOrderForm.Product.value ==
    → "Greek Boat" )
    {
        oOrderForm.Format[0].checked = true;
    }
}
```

  where **Product** is the name of the select box, **Format** is the name of the group of radio buttons and the value of the first ratio button is Slide.

- Finally, to reset all form objects to their initial state, use the reset() method:

```
oFormObject.reset()
```

Project

- Open your previous project file, and save it under the name **chapter_36.html**.
- Remove all content from the body section of the page.
- Copy the file **order_form.html** from the network, and open it using NotePad's File > Open command.
- Copy and paste the entire contents of **order_form.html** into your current project file, into the body element of the page.
- Remove all JavaScript code from the script element in the head section of the page.
- Take some time to open your project file in a browser, and study the code.

- Add an event handler to the form element on your page that will invoke a function called **`check_form()`** when the form is submitted. This function will return **`true`** or **`false`** depending on whether the form passes a number of tests as described below. Remember to precede the event handler's function call with **`return`** to ensure that the form awaits confirmation from the function before proceeding.

- Create the **`check_form()`** function in the head area script element. The function should perform the following steps:

  - If the current value of the selection menu is "":

    - Display an alert with the message "**please select a product**"

    - Give focus to the menu

    - Return **`false`**

  - If no radio buttons are selected:

    - Display an alter with the message "**please specify a format**"

    - Return **`false`**

  - Otherwise, return **`true`**

- Check your work in your browser.

- The **Greek Fishing Boat** is only available in **Slide** format. Add an event handler to the select element which will run a function called **`set_format()`** when its value is changed.

- Create the function **`set_format()`** in the head area script element. The function should check first what the value of the menu is. If it is the **Greek Fishing Boat**, it should then check to see if the **Slide** radio button is checked. If it is not, then the function should correct this.

- Check your work in your browser.

- Modify your check_form() function to check that, if the value of the menu is Greek Fishing Boat, then the Slide radio button is checked. If not, it should report the error as before.

- Finally, note that your **`check_form()`** function is currently not very efficient, in that it will only report one error at a time. This can be tedious for an error prone user, and it is much better practise to observe a form for *all* errors simultaneously.

- Create a variable at the beginning of the function called **`error`**. Set the value of this variable to “”.

- Instead of using an alert box each time an error is found, add the error message to the end of the variable, eg

  ```
  error += “error message\n”;
  ```

  (note the new line code at the end of each message)

- After all checks have been made, if we have caught any errors, the value of **`error`** will no longer be “”. Thus, we can use the following code to report all errors at once:

  ```
  if ( error != “” )
  {
      alert(“The following errors were found: \n\n”
          + error);
      return false;
  }
  else
  {
      return true;
  }
  ```

- Modify your function to be more efficient.

37 Working with Dates

Key Points

- The date object stores all aspects of a date and time from year to milliseconds.
- To use a new date object, you must create it and assign it to an object variable simultaneously. The following code creates a new date object which stores the current date and time and assigns it to the variable **dtNow**

```
dtNow = new Date();
```

- You can specify your own parameters for the date object when you create it:

```
theDate = new Date(
    year, month, day, hours,
    minutes, seconds, mseconds
);
```

- Note that months are represented by the numbers 0 to 11 (January to December). Days are represented by 1 to 31, hours by 0 to 23, minutes and seconds by 0 to 59 and milliseconds by 0 to 999.
- The following code stores 17th July 2004 at 9:15:30pm in the variable **theDate**:

```
theDate = new Date(2004,6,17,21,15,30,0);
```

- The first three parameters are mandatory, while the rest are optional:

```
theDate = new Date(2004,6,17);
```

gives you midnight of the date above.

- You can assign a date to a variable using a string in the following way:

```
theDate =
  new Date("Sun, 10 Oct 2000 21:15:00 – 0500);
```

- Everything from the hours onwards is optional, and in practical terms the day is not necessary either. For example:

```
theDate = new Date("10 Oct 2000");
```

  is a valid date.

Project

- Open your previous project file, and save it under the name **`chapter_37.html`**.

- Remove all content from the body section of the page.

- Create a function in a head section script element that displays an alert box containing the current date and time. Have this function called from a body section script element.

- Now modify your script so that it also writes the date and time 17th July 1955 1:00am to the page in standard date and time format. Do this by entering the appropriate parameters into the **`new Date()`** constructor.

- Now, using parameters once again, add a new line to your script that will display midnight on 17th July 2004 on a separate line under the existing date.

- Finally, using the string approach in the date object, add a new line to your script which will display midnight on 31st December 1999 on a separate line under the existing information and in standard date and time format.

38 Retrieving Information from Date Objects

Key Points

- Below is a selection of some useful methods which enable you to retrieve some useful information from date objects:

Method	Returns
getDate()	The day of the month (1-31)
getDay()	The day of the week (0 = Sunday, 6 = Saturday)
getFullYear()	The year as four digits
getHours()	The hour (0-23)
getMilliseconds()	The milliseconds (0-999)
getMinutes()	The minutes (0-59)
getMonth()	The month (0-11)
getSeconds()	The seconds (0-59)
getTime()	The date and time in milliseconds – also called Unix Time

- JavaScript stores date and time information internally as the number of milliseconds from 1st January 1970. This is common to most programming languages, and is actually the way most computer systems store time information.

- For example:

```
dtNow = new Date();
document.write(dtNow.getTime());
```

  writes the number of milliseconds that have passed since 1st January 1970.

- Being aware that there are 1000 milliseconds in a second, 60 seconds in a minute, 60 minutes in an hour and 24 hours in a day, you can establish the number of days (or hours or minutes etc) between two dates by subtracting on date in millisecond format from the other in millisecond format. To achieve the units you require, perform the appropriate division (so for the number of minutes, divide the result of the subtraction first by 1000, then by 60), and then use Math.floor() on the result to round the number down as required.

Project

- Open your previous project file, and save it under the name **`chapter_38.html`**.

- Clear any JavaScript from the page's script elements.

- Create a function in the head section script element that performs the following:

  - Create two arrays, one storing the days of the week ("Sunday", "Monday" etc), the other storing the months of the year ("January", "February" etc)

  - Use a new date object, along with the arrays and the appropriate date methods to write today's date to the page in the following format:

    Today it is: dayName, month, dayNumber.

    for example:

    Today it is Tuesday, October 17.

- Below the date, write the time in the following format:

  It is currently hh:mm am (or pm)

  Note: *to convert from the 24hr clock, subtract 12 from any value over 12, and replace a zero value with 12. Anything greater than or equal to twelve should receive a "pm" suffix.*

- Below the time, write the number of days since the start of the millennium in the following format:

  It is n days since the start of the millennium.

- Finally, call your function from the body area script element on your page, and check your work in your browser.

39 Creating a JavaScript Clock

Key Points

- **`setTimeout()`** is a method of the window object. In its common form, it enables you to run any JavaScript function after an allotted time (in milliseconds) has passed. For example:

  ```
  window.setTimeout("functionName()", 1000)
  ```

  will run **`functionName()`** after one second. Note the quote marks!

- You can have any number of "timed out" methods running at any one time. To identify each "timeout", it is a good idea to assign the return value of each one you create to an object variable:

  ```
  firstTimeout =
    window.setTimeout("functionName()", 1000)
  ```

- The most common reason to track timeouts is to be able to cancel them if necessary – for example, you may wish a window to close after 5 seconds unless a button is clicked:

  ```
  firstTimeout =
     window.setTimeout("window.close()", 5000);
  ```

  adding the following code to the button's **`onclick`** event handler:

  ```
  window.clearTimeout(firstTimeout);
  ```

  will do the trick.

- Ordinarily, it is a bad idea to "recurse" a function – to have a function call itself. For example, imagine the annoyance of the following:

```
function annoy_me( )
{
    window.alert("BOO!");
    annoy_me();
}
```

  This will potentially keep popping up alert boxes, and perhaps even lock up the user's computer!

- However, you may wish a function to use a **`setTimeout`** method to call itself periodically. Imagine a function that calls itself every second to check on the time, and then displays the result in the same place. In effect, this could be seen as a digital clock.

- If you want to clear the clock's timeout – which in this case would stop the clock – the timeout must already be in operation. You can't clear a timeout that doesn't yet exist. So, you have to check first whether a timeout is in operation. One way of doing this is to set a Boolean variable to true whenever the clock is started. By checking this variable before you attempt to clear the timeout, you know whether or not the clock is running and therefore can be stopped.

- To show an am/pm clock you obviously need to carry out the conversion you created in the previous chapter's project.

- But, so as the display doesn't show a "moving" effect at different times, you need to consider the situation where either the minutes or seconds on the clock are fewer than 10. For example, consider:

```
10:59:59 am
```

  changing to:

```
11:0:0 am
```

- In this situation, we need to concatenate an extra "0" on to the front of the actual value to produce:

  11:00:00 am

  as we might expect.

- We can use a shortened form of the if conditional to make this simple. For example:

  s = theDate.getSeconds;
  s = (s < 10) ? "0" + s : s;

- What is happening here? First, we get the current value of the seconds and store that in a variable called "s". Next, we reset the value of s depending on the condition in the brackets. The prototype of this form of the if conditional is:

  ```
  var = ( condition )
      ? value if true
      : value if false;
  ```

  in other words, we can think of the ? as being like the opening brace { of an if conditional, the : as being the } else {, and the ; as being the closing brace. The result of the conditional test is then stored in var.

  Note that this has been split over three lines to aid reading. In practice (see above) we can place this on one line for ease of use.

Project

- Open your previous project file, and save it under the name **chapter_39.html**.

- Remove all content from the body section of the page.

- In the body section, create a form input element with the id **clockBox**, and two form input buttons labelled Start and Stop.

- In the head section scrip element, create two empty functions – **start_clock()** and **stop_clock().**

- Before the two function definitions, as the first statement of the script element, create a variable called **timer**, and assign it a value of **null**.

- Immediately below that statement, create a variable called timer_running. Assign it a value of false.
- We will use these variables to track the clock's status.
- Now, add statements to the start_clock() function that will:
  - Get the current time.
  - Format the current time as hh:mm:ss am/pm (as appropriate).
  - Display the formatted time in the text field on the page.
  - Create a timeout which will run the **`start_clock()`** function again in half a second, and assign that timeout's return object to the **`timer`** object variable.
  - Set the **`timer_running`** variable to **`true`**.
- Add statements to the **`stop_clock()`** function that will:
  - Check to see if the clock is running.
  - If it is, clear the timer timeout and set **`timer_running`** to **`false`**.
- Finally, add event handlers to the two form buttons to ensure that the one labelled Stop calls the function **`stop_clock()`** when clicked, and the one labelled Start calls the function **`start_clock()`** when clicked.
- While this should work (test your code!), you'll notice that the output is a little ugly. Displaying the time in a text field is not the most unobtrusive way to show a clock on a page.
- Luckily, we can modify our code very slightly to achieve something much more professional.
- Replace your input field with the following:

  ```
  <span id="clockBox"></span>
  ```

- **`<span>`** is an HTML tag that allows you to mark areas of content without any semantic meaning – eg, the clock is not a paragraph or a header, so we don't want to label it as such.

- With JavaScript, we can control the content of just about any element on the page. In our previous example, we used **`getElementById`** (hopefully!) to obtain a reference to the input field, and then altered its **`value`** property to show our clock.

- Since we are using the same id value here, we do not have to alter our function too much. However, the **`getElementById`** method now returns a different type of object – one that has no **`value`** property.

- However, all content tags in HTML (like **`<p>`**, **`<h1>`**, **`<div>`** etc) have a special property when they are returned as JavaScript objects which refers to their text content – the property is **`innerHTML`**.

- We can use this to alter the content of the tags. For example, if our previous input-field solution had the following code:

  ```
  clk = document.getElementById("clockBox");
  clk.value = formatted_time;
  ```

  where **`formatted_time`** is a variable containing the formatted time as required, then replacing it with this code:

  ```
  clk = document.getElementById("clockBox");
  clk.innerHTML = formatted_time;
  ```

  will allow us to use the modified **`<span>`** element in place of the **`<input>`** element.

- Try adapting your code to use this method of displaying content on the page.