**Javascript**

In the beginning of the Web, there was HTML and the Common Gateway Interface (CGI). HTML defines the parts of a text document and instructs the user agent (usually the web browser) how to show it—for example, text surrounded by the tags <p></p> becomes a paragraph. Within that paragraph you may have <h1></h1> tags that define the main page heading. Notice that for most opening tags, there is a corresponding closing tag that begins with </. HTML has one disadvantage—it has a fixed state. If you want to change something, or use data the visitor entered, you need to make a round-trip to a server. Using a **dynamic technology** (such as ColdFusion, ASP, ASP.NET, PHP, or JSP) you send the information from forms, or from parameters, to a server, which then performs calculating/testing/database lookups, etc. The application server associated with these technologies then writes an HTML document to show the results, and the resulting HTML document is returned to the browser for viewing. The problem with that is it means every time there is a change, the entire process must be repeated (and the page reloaded). This is cumbersome, slow, and not as impressive as the new media “Internet” promised us to be. It is true that at least the Western world has the benefit of fast Internet connections these days, but displaying a page still means a reload, which could be a slow process that frequently fails (ever get an Error 404?).

We need something slicker—something that allows web developers to give immediate feedback to the user and change HTML without reloading the page from the server. Just imagine a form that needs to be reloaded every time there’s an error in one of its fields—isn’t it handier when something flags the errors immediately, without needing to reload the page from the web server? This is one example of what JavaScript can do for you. Some information, such as calculations and verifying the information on a form, may not need to come from the server. JavaScript is executed by the user agent (normally a browser) on the visitor’s computer. We call this **client-side code**. This could result in fewer trips to the server and faster-running web sites.

**What Is JavaScript?**

JavaScript started life as **LiveScript**, but Netscape changed the name—possibly because of the excitement being generated by Java—to JavaScript. The name is confusing though, as there is no real connection between Java and JavaScript—although some of the syntax looks similar.

***“Java is to JavaScript what Car is to Carpet”***

Netscape created the JavaScript language in 1996 and included it in their Netscape Navigator (NN) 2.0 browser via an interpreter that read and executed the JavaScript added to .html pages. The language has steadily grown in popularity since then, and is now supported by the most popular browsers. The good news is that this means JavaScript can be used in web pages for all major modern browsers. The not-quite-so-good news is that there are differences in the way the different browsers implement JavaScript, although the core JavaScript language is much the same. However, JavaScript can be turned off by the user—and many companies and other institutions require their users to do so for security reasons. The great thing about JavaScript is that once you’ve learned how to use it for browser programming, you can move on to use it in other areas. Microsoft’s server—IIS—uses JavaScript to program server-side web pages (ASP), PDF files now use JavaScript, and even Windows administration tasks can be automated with JavaScript code. A lot of applications such as Dreamweaver and Photoshop are scriptable with JavaScript. Operating system add-ons like the Apple Dashboard or Konfabulator on Linux and Windows even allow you to write small helper applications in JavaScript.

Lately a lot of large companies also offer application programming interfaces (APIs) that feature JavaScript objects and methods you can use in your own pages—Google Maps being one of them. You can offer a zoomable and scrollable map in your web site with just a few linesof code.

Even better is the fact that JavaScript is a lot easier to develop than higher programming languages or server-side scripting languages. It does not need any compilation like Java or C++, or to be run on a server or command line like Perl, PHP, or Ruby: all you need to write, execute, debug, and apply JavaScript is a text editor and a browser—both of which are supplied with any operating system. There are, of course, tools that make it a lot easier for you, examples being JavaScript debuggers like Mozilla Venkman, Microsoft Script Debugger.

**Problems and Merits of JavaScript**

JavaScript has been an integral part of web development over the last few years, but it has also been used wrongly. As a result, it has gotten a bad reputation. The reason for this is gratuitous JavaScript effects, like moving page elements and pop-up windows, which might have been impressive the first time you saw them but soon turned out to be just a “nice to have” and in some cases even a “nice to not have any longer.” A lot of this comes from the days of **DHTML**. The term **user agent** and the lack of understanding what a user agent is can also be a problem. Normally, the user agent is a browser like Microsoft Internet Explorer (MSIE), Netscape, Mozilla (Moz), Firefox (Fx), Opera, or Safari. However, browsers are not the only user agents on the Web. Others include

• Assistive technology that helps users to overcome the limitations of a disability—like text-to-speech software or Braille displays

• Text-only agents like Lynx

• Web-enabled applications

• Game consoles

• Mobile/cell phones

• PDAs

• Interactive TV set-top boxes

• Search engines and other indexing programs

• And many more

This large variety of user agents, of different technical finesse (and old user agents that don’t get updated), is also a great danger for JavaScript. Not all visitors to your web site will experience the JavaScript enhancements you applied to it. A lot of them will also have JavaScript turned off—for security reasons. **JavaScript can be used for good and for evil. If the operating system—like unpatched Windows—allows you to, you can install viruses or Trojan Horses on a computer via JavaScript or read out user information and send it to another server**. In many cases, you might want to have a server-side backup plan. It would test to see whether the user agent supports the functionality desired and, if it doesn’t, the server takes over. Independence of scripting languages is a legal requirement for web sites. This means that if the site you developed cannot be used without JavaScript, or your JavaScript enhancements are expecting a certain ability of the users or their user agent without a fallback, your client could be sued for discrimination. However, JavaScript is not evil or useless, and it is a great tool to help your visitor to surf web sites that are a lot slicker and less time-consuming.

**Why Use JavaScript If It Cannot Be Relied On?**

Just because it may not always be available doesn’t mean that JavaScript shouldn’t be used at all. It should simply not be the only means of user interaction.

**The merits of using JavaScript are**

• **Less server interaction**: You can validate user input before sending the page off to the server. This saves server traffic, which means saving money.

• **Immediate feedback to the visitors**: They don’t have to wait for a page reload to see if they have forgotten to enter something

• **Automated fixing of minor errors**: For example, if you have a database system that expects a date in the format *dd-mm-yyyy* and the visitor enters it in the form *dd/mm/yyyy*, a clever JavaScript script could change this minor mistake prior to sending the form to the server. If that was the only mistake the visitor made, you can save her an error message— thus making it less frustrating to use the site.

• **Increased usability by allowing visitors to change and interact with the user interface**

**without reloading the page**: For example, by collapsing and expanding sections of the page or offering extra options for visitors with JavaScript. A classic example of this would be select boxes that allow immediate filtering, such as only showing the available destinations for a certain airport, without making you reload the page and wait for the result.

• **Increased interactivity**: You can create interfaces that react when the user hovers over them with a mouse or activates them via the keyboard. This is partly possible with CSS and HTML as well, but JavaScript offers you a lot wider—and more widely supported— range of options.

• **Richer interfaces**: If your users allow for it, you can use JavaScript to include such items as drag-and-drop components and sliders—something that originally was only possible in thick client applications your users had to install, such as Java applets or browser plug-ins like Flash.

**Lightweight environment**: Instead of downloading a large file like a Java applet or a Flash movie, scripts are small in file size and get cached (held in memory) once they have been loaded. JavaScript also uses the browser controls for functionality rather than its own user interfaces like Flash or Java applets do. This makes it easier for users, as they already know these controls and how to use them. Modern Flash and Macromedia Flex applications do have the option to stream media and—being vector based—are visually scalable, something JavaScript and HTML controls aren’t. On the other hand, they require the plug-in to be installed.

**Objects**

Objects are central to the way we use JavaScript. Objects in JavaScript are in many ways like objects in the world outside programming (it does exist, I just had a look). In the real world, an object is just a “thing” (many books about object-oriented programming compare objects to nouns): a car, a table, a chair, and the keyboard I’m typing on. Objects have

**Properties** (analogous to adjectives): The car is *red*.

**Methods** (like verbs in a sentence): The method for starting the car might be *turn ignition key*.

**Events**: Turning the ignition key results in the *car starting* event.

Object Oriented Programming (OOP) tries to make programming easier by modeling realworld objects. Let’s say we were creating a car simulator. First, we would create a car object, giving it properties like *color* and *current speed*. Then we’d need to create methods: perhaps a *start* method to start the car, and a *break* method to slow the car, into which we’d need to pass information about how hard the brakes should be pressed so that we can determine the slowing effect. Finally, we would want some events, for example, a *gasoline low* event to remind us to fill up the car.

Object-oriented programming works with these concepts. This way of designing software is now very commonplace and influences many areas of programming—but most importantly to us, it’s central to JavaScript and web browser programming. Some of the objects we’ll be using are part of the language specification: the String object, the Date object, and the Math object, for example. The same objects would be available to JavaScript in a PDF file and on a web server. These objects provide lots of useful functionality that could save us tons of programming time. The Date object, for example, allows you to

obtain the current date and time from the client (such as a user’s PC). It stores the date and provides lots of useful date-related functions, for example, converting the date/time from one time zone to another. These objects are usually referred to as **core objects**, as they are independent of the implementation. The browser also makes itself available for programming through objects that allow us to obtain information about the browser and to change the look and feel of the application. For example, the browser makes available the Document object, which represents a web page available to JavaScript. We can use this in JavaScript to add new HTML to the web page being viewed by the user of the web browser. If you were to use JavaScript with a different host, with a Windows server for example, then you’d find that the server hosting JavaScript exposes a very different set of host objects, their functionality being related to things you want to do on a web server. You’ll also see in Chapter 3 that JavaScript allows us to create our own objects. This is a powerful feature that allows us to model real-world problems using JavaScript. To create a new object, we need to specify the properties and methods it should have using a template called a **class**. A class is a bit like an architect’s drawing in that it specifies what should go where and do what, but it doesn’t actually create the object.

**Data, Data Types, and Data Operators**

Data is used to store information, and, in order to do that more effectively, JavaScript needs to have each piece of data assigned a **type**. This type stipulates what can or cannot be done with the data. For example, one of the JavaScript data types is **number**, which allows you to perform certain calculations on the data that it holds. The three most basic data types that store data in JavaScript are

• **String**: A series of characters, for example, “some characters”

• **Number**: A number, including floating point numbers

• **Boolean**: Can contain a true or false value

These are sometimes referred to as **primitive** data types in that they store only single values. There are two slightly different primitive data types as well. These don’t store information, but instead warn us about a particular situation:

• **Null**: Indicates that there is no data.

• **Undefined**: Indicates that something has not been defined and given a value. This is important when you’re working with variables.

**The String Data Type**

The JavaScript interpreter expects string data to be enclosed within single or double quotation marks (known as **delimiters**). The quotation marks won’t be written out to the page because they are not part of the string; they simply tell JavaScript where the string starts and ends. Both methods are fine, just as long as you close the string the same way you opened it and don’t try to delimit it like this:

**document.write( 'some characters" );**

**document.write( "some characters' );**

Of course, you might want to use a single or double quotation mark inside the string itself, in which case you need to use a distinct delimiter. If you used double quotation marks, the instructions will be interpreted as you intended:

**document.write( "Paul's characters " );**

But if you used single quotations marks, they won’t be:

**document.write( 'Paul's characters' );**

This will give you a syntax error because the JavaScript interpreter thinks the string ends after the *l* in *Paul* and doesn’t understand what is happening afterwards.

You can avoid creating JavaScript syntax errors like this one by using single quotation marks to delimit any string containing double quotes and vice versa:

**document.write( "Paul's numbers are 123" );**

**document.write( 'some "characters"' );**

If, on the other hand, you wanted to use both single and double quotation marks in your string, you need to use something called an **escape sequence**. In fact, it’s better coding practice to use escape sequences instead of the quotation marks we’ve been using so far, because they make your code easier to read. Escape Sequences.

Let’s amend this string, which causes a syntax error:

**document.write( 'Paul's characters' );** so that it uses the escape sequence (\') and is correctly interpreted:

**document.write( 'Paul\'s characters' );**

The escape sequence tells the JavaScript interpreter that the single quotation mark belongs to

the string itself and isn’t a delimiter.

**JavaScript Variables**

JavaScript is probably the most forgiving language when it comes to variables. You don’t need to define what a variable is before you can use it, and you can change the type of a variable any time in the script. However, to ease maintenance and keep up a stricter coding syntax, it is a good idea to declare variables explicitly at the beginning of your script or—in the case of local variables—your function We declare a variable by giving it a unique name and using the **var** keyword. Variable names have to start with a letter of the alphabet or with an underscore, while the rest of the name can be made up only of numbers, letters, the dollar sign ($), and underscore characters. Do not use any other characters. Like most things in JavaScript, variable names are case sensitive: thisVariable and ThisVariable are different variables. Be very careful about naming your variables; you can run into all sorts of trouble if you don’t name them consistently. To that end, most programmers use **camel notation**: the name of the variable begins with a lowercase letter while subsequent words are capitalized and run in without spaces. Thus the name thisVariable. Always give your variables meaningful names. In the next example we’ll build, we’re

going to write an exchange rate conversion program, so we’ll use variable names like euroToDollarRate and dollarToPound. There are two advantages to naming variables descriptively: it’s easier to remember what the code is doing if you come back to it at a later date, and it’s easier for someone new to the code to see what’s going on. Code readability and layout are very important to the development of web pages. It makes it quicker and easier to spot errors and debug them, and to amend the code as you want to. The prompt() function is a JavaScript function that asks the user to enter a value and then returns it to the code. Here we’re assigning the value entered to the variable eurosToConvert. Initializing your variables is a very good idea, especially if you can give them a default value that’s useful to the application. Even initializing a variable to an empty string can be a good idea, because you can check back on it without bringing up the error messages that would have popped up if it didn’t have a value.

**Converting Different Types of Data**

For the most part, the JavaScript interpreter can work out what data types we want to be used. In the following code, for example, the interpreter understands the numbers 1 and 2 to be of

number data type and treats them accordingly: This will write string into the page. The way to ensure that the interpreter is using the desired number data type is to **explicitly** declare that the data is a number. There are three functions you can use to do this:

**• Number():** Tries to convert the value of the variable inside the parentheses into a number.

**• parseFloat():** Tries to convert the value to a floating point. It parses the string character

by character from left to right, until it encounters a character that can’t be used in a number. It then stops at that point and evaluates this string as a number. If the first character can’t be used in a number, the result is **NaN** (which stands for **Not a Number**).

**• parseInt():** Converts the value to an integer by removing any fractional part without rounding the number up or down. Anything nonnumerical passed to the function will be discarded. If the first character is not +, -, or a digit, the result is NaN.

**Objects JavaScript Supplies You with: String, Date, and Math**

These three objects do three different things:

• String **object**: Stores a string, and provides properties and methods for working

with strings

• Date **object**: Stores a date, and provides methods for working with it

• Math **object**: Doesn’t store data, but provides properties and methods for manipulating mathematical data

Let’s start with the String object.

The **String** Object

Earlier we created string primitives by giving them some characters to hold, like this:

var myPrimitiveString = "ABC123";

A String **object** does things slightly differently, not only allowing us to store characters, but also providing a way to manipulate and change those characters. You can create String objects explicitly or implicitly. We can still use the String object’s methods on it though. JavaScript will simply convert the string primitive to a temporary String object, use the method on it, and then change the data type back to string. We can try that out using the length **property** of the String object:

We can also create String objects explicitly, using the new keyword together with the String() **constructor**:

The only difference between this script and the previous one is in the first line where we create the new object and supply some characters for the String object to store:

**var myStringObject = new String( "abc" );**

The result of checking the length property is the same whether we create the String object implicitly or explicitly. The only real difference between creating String objects explicitly or implicitly is that creating them explicitly is marginally more efficient if you’re going to be using the same String object again and again. Explicitly creating String objects also helps prevent the JavaScript interpreter getting confused between numbers and strings, as it can do.

**Using the String Object’s Methods**

The String object has a lot of methods, so we’ll limit our discussion to two of them here, the indexOf() and substring() methods. JavaScript strings, as you’ve seen, are made up of characters. Each of these characters is given an index. The index is zero-based, so the first character’s position has the index 0, the second 1, and so on. The method indexOf() finds and returns the position in the index at which a substring begins (and the lastIndexOf() method returns the position at which the substring ends). For example, if we want our user to enter an e-mail address, we could check that they’d included the @ symbol in their entry. (While this wouldn’t ensure that the address is valid, it would at least go some way in that direction. We’ll be working with much more complex data checking later on in the book.) Let’s do that next, using the prompt() method to obtain the user’s e-mail address and then check the input for the @ symbol, returning the index of the symbol using indexOf():

The substring() method carves one string from another string, taking the indexes of the start and end position of the substring as parameters. We can return everything from the first index to the end of the string by leaving off the second parameter. So to extract all the characters from the third character (at index 2) to the sixth character (index 5), we’d write

<html>

<body>

<script type="text/javascript">

var myOldString = "Hello World";

var myNewString = myOldString.substring( 2, 5 );

document.write( myNewString );

</script>

</body>

</html>

You should see Hello written out to the browser. Note that the substring() method copies the substring that it returns, and it doesn’t alter the original string. The substring() method really comes into its own when you’re working with unknown values.

**The Date Object**

JavaScript doesn’t have a primitive date data type, so we can only create Date objects explicitly. We create new Date objects the same way as we create String objects, using the new keyword together with the Date() constructor. This line creates a Date object containing the current date and time:

var todaysDate = new Date();

To create a Date object that stores a specific date or time, we simply put the date, or date and time, inside the parentheses:

var newMillennium = new Date( "1 Jan 2000 10:24:00" );

Different countries describe dates in a different order. For example, in the US dates are specified in *MM/DD/YY*, while in Europe they are *DD/MM/YY*, and in China they are *YY/MM/DD*. If you specify the month using the abbreviated name, then you can use any order:

var someDate = new Date( "10 Jan 2002" );

var someDate = new Date( "Jan 10 2002" );

var someDate = new Date( "2002 10 Jan" );

Although we usually think of month 9 as September, JavaScript starts counting months from 0 (January), and so September is represented as month 8.

**Using the Date Object**

The Date object has a lot of methods that you can use to get or set a date or time. You can use local time (the time on your computer in your time zone) or UTC (Coordinated Universal Time, once called Greenwich Mean Time). While this can be very useful, you need to be aware when you’re working with Date that many people don’t set their time zone correctly. Let’s look at an example that demonstrates some of the methods:

<html>

<body>

<script type="text/javascript">

// Create a new date object

var someDate = new Date( "31 Jan 2003 11:59" );

// Retrieve the first four values using the

// appropriate get methods

document.write( "Minutes = " + someDate.getMinutes() + "<br>" );

document.write( "Year = " + someDate.getFullYear() + "<br>" );

document.write( "Month = " + someDate.getMonth() + "<br>" );

document.write( "Date = " + someDate.getDate() + "<br>" );

// Set the minutes to 34

someDate.setMinutes( 34 );

document.write( "Minutes = " + someDate.getMinutes() + "<br>" );

// Reset the date

someDate.setDate( 32 );

document.write( "Date = " + someDate.getDate() + "<br>" );

document.write( "Month = " + someDate.getMonth() + "<br>" );

</script>

</body>

</html>

Here’s what you should get:

Minutes = 59

Year = 2003

Month = 0

Date = 31

Minutes = 34

Date = 1

Month = 1

JavaScript knows that there aren’t 32 days in January, so instead of trying to set the date to the

January 32, the interpreter counts 32 days beginning with January 1, which gives us February 1.

The **Math** Object

The Math object provides us with lots of mathematical functionality, like finding the square of a number or producing a random number. The Math object is different from the Date and String objects in two ways:

• You can’t create a Math object explicitly, you just go ahead and use it.

• The Math object doesn’t store data, unlike the String and Date object.

You call the methods of the Math object using the format:

Math.methodOfMathObject( aNumber ):

alert( "The value of pi is " + Math.PI );

We’ll look at a few of the commonly used methods next (you can find a complete reference by running a search at http://www.mozilla.org/docs/web-developer/). We’ll look at the methods for rounding numbers and generating random numbers here.

**Rounding Numbers**

You saw earlier that the parseInt() function will make a fractional number whole by removing everything after the decimal point (so 24.999 becomes 24). Pretty often you’ll want more mathematically accurate calculations, if you’re working with financial calculations, for example, and for these you can use one of the Math object’s three rounding functions: round(), ceil(), and floor(). This is how they work:

• round(): Rounds a number up when the decimal is . 5 or greater

• ceil() **(as in ceiling)**: Always rounds up, so 23.75 becomes 24, as does 23.25

• floor(): Always rounds down, so 23.75 becomes 23, as does 23.25

**Generating a Random Number**

You can generate a fractional random number that is 0 or greater but smaller than 1 using the Math object’s random() method. Usually you’ll need to multiply the number, and then use one

of the rounding methods in order to make it useful.

For example, in order to mimic a die throw, we’d need to generate a random number between 1 and 6. We could create this by multiplying the random fraction by 5, to give a fractional number between 0 and 5, and then round the number up or down to a whole number using the round() method. (We couldn’t just multiply by 6 and then round up every time using ceil(), because that would give us the occasional 0.) Then we’d have a whole number between 0 and 5, so by adding 1, we can get a number between 1 and 6. This approach won’t give you a perfectly balanced die, but it’s good enough for most purposes. Here’s the code:

<html>

<body>

<script type="text/javascript">

var diceThrow = Math.round( Math.random( ) \* 5 ) + 1;

document.write( "You threw a " + diceThrow );

</script>

</body>

</html>

**Arrays**

JavaScript allows us to store and access related data using an **array**. An array is a bit like a row of boxes (**elements**), each box containing a single item of data. An array can work with any of the data types that JavaScript supports. So, for example, you could use an array to work with a list of items that the users will select from, or for a set of graph coordinates, or to reference a group of images. Array objects, like String and Date objects, are created using the new keyword together with the constructor. We can initialize an Array object when we create it:

var preInitArray = new Array( "First item", "Second item","Third Item" );

Or

set it to hold a certain number of items:

var preDeterminedSizeArray = new Array( 3 );

Or

just create an empty array:

var anArray = new Array();

**You can add new items to an array by assigning values to the elements:**

anArray[0] = "anItem"

anArray[1] = "anotherItem"

anArray[2] = "andAnother"

**Once we’ve populated an array, we can access its elements through their indexes or positions (which, once again, are zero-based) using square brackets:**

<html>

<body>

<script type="text/javascript">

var preInitArray = new Array( "First Item","Second Item", "Third Item" );

document.write( preInitArray[0] + "<br>" );

document.write( preInitArray[1] + "<br>" );

document.write( preInitArray[2] + "<br>" );

</script>

</body>

</html>

Using index numbers to store items is useful if you want to loop through the array—we’ll

be looking at loops next.

**The Array Object’s Methods and Properties**

One of the most commonly used properties of the Array object is the length property, which returns the index one count higher than the index of the last array item in the array. If, for example, you’re working with an array with elements with indexes of 0, 1, 2, 3, the length will be 4—which is useful to know if you want to add another element. The Array object provides a number of methods for manipulating arrays, including methods for cutting a number of items from an array, or joining two arrays together. We’ll look at the methods for concatenating, slicing, and sorting next.

**Cutting a Slice of an Array**

The **slice()** method is to an Array object what the substring() method is to a String object. You simply tell the method which elements you want to be sliced. This would be useful, for example, if you wanted to slice information being passed using a URL. The slice() method takes two parameters: the index of the first element of the slice, which will be included in the slice, and the index of the final element, which won’t be. To access the second, third, and fourth values from an array holding five values in all, we use the indexes 1 and 4:

<html>

<body>

<script type="text/javascript">

// Create and initialize the array

var fullArray = new Array( "One", "Two", "Three","Four", "Five" );

// Slice from element 1 to element 4 and store

// in new variable sliceOfArray

var sliceOfArray = fullArray.slice( 1, 4 );

// Write out new ( zero-based ) array of 3 elements

document.write( sliceOfArray[0] + "<br>" );

document.write( sliceOfArray[1] + "<br>" );

document.write( sliceOfArray[2] + "<br>" );

</script>

</body>

</html>

The new array stores the numbers in a new zero-based array, so slicing indexes 0, 1, and 2

gives us the following:

Two

Three

Four

**Joining Two Arrays**

The Array object’s concat() method allows us to concatenate arrays. We can add two or more arrays using this method, each new array starting where the previous one ends. Here we’re joining three arrays: arrayOne, arrayTwo, and arrayThree:

<html>

<body>

<script type="text/javascript">

var arrayOne = new Array( "One", "Two", "Three","Four", "Five" );

var arrayTwo = new Array( "ABC", "DEF", "GHI" );

var arrayThree = new Array( "John", "Paul", "George","Ringo" );

var joinedArray = arrayOne.concat( arrayTwo, arrayThree );

document.write( "joinedArray has " + joinedArray.length + ➥

" elements<br>" );

document.write( joinedArray[0] + "<br>" )

document.write( joinedArray[11] + "<br>" )

</script>

</body>

</html>

The new array, joinedArray, has 12 items. The items in this array are the same as they were in each of the previous arrays; they’ve simply been concatenated together. The original arrays remain untouched.

Sorting an Array

The sort() method allows us to sort the items in an array into alphabetical or numerical order:

<html>

<body>

<script type="text/javascript">

var arrayToSort = new Array( "Cabbage", "Lemon","Apple", "Pear", "Banana" );

var sortedArray = arrayToSort.sort( );

document.write( sortedArray[0] + "<br>" );

document.write( sortedArray[1] + "<br>" );

document.write( sortedArray[2] + "<br>" );

document.write( sortedArray[3] + "<br>" );

document.write( sortedArray[4] + "<br>" );

</script>

</body>

</html>

The items are arranged like this:

Apple

Banana

Cabbage

Lemon

Pear

If, however, you lower the case of one of the letters—the *A* of *Apple*, for example—then you’ll end up with a very different result. The sorting is strictly mathematical—by the number of the character in the ASCII set, not like a human being would sort the words. If you wanted to change the order the sorted elements are displayed, you can use the reverse() method to display the last in the alphabet as the first element:

<script type="text/javascript">

var arrayToSort = new Array( "Cabbage", "Lemon", "Apple", "Pear", "Banana" );

var sortedArray = arrayToSort.sort( );

var reverseArray = sortedArray.reverse( );

document.write( reverseArray[0] + "<br />" );

document.write( reverseArray[1] + "<br />" );

document.write( reverseArray[2] + "<br />" );

document.write( reverseArray[3] + "<br />" );

document.write( reverseArray[4] + "<br />" );

</script>

The resulting list is now in reverse order:

Pear

Lemon

Cabbage

Banana

Apple

**The JavaScript *window object*** sits at the top of the JavaScript Object hierarchy and represents the browser window (or windows if you have more than one browser window open at any one time). Up until this chapter we have focused on the internals and syntax of JavaScript. In this chapter we will begin to make things happen on the screen (which, after all, is one of the main purposes of JavaScript). The window object allows developers to perform tasks such as opening and closing browser windows, displaying alert and prompt dialogs and setting up timeouts (specifying an action to take place after a specified period of time).

|  |  |
| --- | --- |
| **Events** | **Description** |
| onblur | Fires when the window loses focus. |
| onerror | Fires when a JavaScript error occurs.  Window.onerror=function(){  } |
| onfocus | Fires when the focus is set on the current window. |
| onload | Fires when the page has finished loading, including images. This is a popular event to use to run some JavaScript once everything on the page has loaded/ is available:  window.onload=function(){ } |
| onresize | Fires when the window is resized. |
| onscroll | Fires when the window is scrolled.  window.onscroll=function(){ } |
| onbeforeunload |  |
| onunload | Fires when the page is unloaded- process cannot be overruled at this point. Often used to run code cleanup routines. |

**Properties**

|  |  |
| --- | --- |
| **Properties** | **Description** |
| closed | Returns the Boolean variable indicating whether window has been closed or not. |
| defaultStatus | Read/write property that reflects the default window status bar message that appears. |
| [document](http://www.javascriptkit.com/jsref/document.shtml) | Reference to the current document object. |
| frames | An array referencing all of the frames in the current window, including IFRAME elements. Useframes.length to probe the number of frames. |
| history | Reference to the History object of JavaScript, which contains information on the URLs the visitor has visited within the window. |
| length | Returns the number of frames contained in the window, which includes IFRAMEs. |
| location | Returns or sets the location object of JavaScript, which contains information on the current URL. |
| name | The name of the window as optionally specified when calling window.open(). |
| opener | Contains a reference to the window that opened the secondary window via window.open(). This property should be invoked in the secondary window.. |
| parent | Reference to the parent frameset window of the current window, assuming current window is a frame. Otherwise, it simply refers to current window. |
| self | A synonym for the current window. |
| status | A read/write property that allows you to probe and write to the browser's status bar. |
| top | A synonym for the topmost browser window. |
| window | References the current window. Same as "self." |

|  |  |
| --- | --- |
| **Methods** | **Description** |
| **alert(msg)** | Displays an Alert dialog box with the desired message and OK button. |
| blur() | Removes focus from the window in question, sending the window to the background on the user's desktop. |
| clearInterval(ID) | Clears the timer set using var ID=setInterval(). |
| clearTimeout(ID) | Clears the timer set using var ID=setTimeout(). |
| **close()** | Closes a window. |
| **confirm(msg)** | Displays a Confirm dialog box with the specified message and OK and Cancel buttons. Returns either true or false, depending on which button the user has clicked on, for example:  var yourstate=window.confirm("Are you sure you are ok?") if (yourstate) //Boolean variable. Sets to true if user pressed "OK" versus "Cancel."  window.alert("Good!") |
| focus() | Sets focus to the window, bringing it to the forefront on the desktop. |
| moveBy(dx, dy) | Moves a window by the specified amount in pixels. |
| moveTo(x, y) | Moves a window to the specified coordinate values, in pixels. The following opens a window and centers it on the user's screen: |
| **open(URL, [name], [features], [replace])** | Opens a new browser window. "Name" argument specifies a name that you can use in the target attribute of your <a> tag. |
| print() | Prints the contents of the window or frame. |
| **prompt(msg, [input])** | Displays a Prompt dialog box with a message. Optional "input" argument allows you to specify the default input (response) that gets entered into the dialog box. This function returns the string the user has entered:  var yourname=window.prompt("please enter your name")  alert(yourname) |
| resizeBy(dx, dy) | Resizes a window by the specified amount in pixels. |
| resizeTo(x y) | Resizes a window to the specified pixel values. The following will resize the current window to be maximized on the screen in browsers with no security hang ups over such an operation (IE does and will do nothing):  window.resizeTo(screen.availWidth, screen.availHeight) //see also the screen object |
| scrollBy(dx, dy) | Scrolls a window by the specified amount in pixels. |
| scrollTo(x, y) | Scrolls a window to the specified pixel values. |
| stop() | Stops the window from loading. NS4/NS6+ exclusive method. Use the optional "args" to pass any number of arguments to the function. |

Let us take examples of some method of window object as alert(), prompt() and open().

<html>

<head>

<title>Java Script </title>

<script>

function disp()

{

alert("Alert box is example of window object method");

}

</script>

</head>

<body>

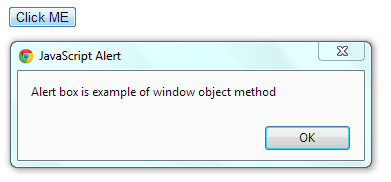
<form>

<input type="button" value="Click ME" onclick="disp();">

</form>

</body>

</html>



<html>

<head>

<script>

arr1=new Array(10);

for(var i=0;i<5;i++)

{

arr1[i]=prompt("Enter Fruit Name");

}

for(var i=0;i<5;i++)

{

document.write(arr1[i]+"<br>");

}

</script>

</head>

<body>

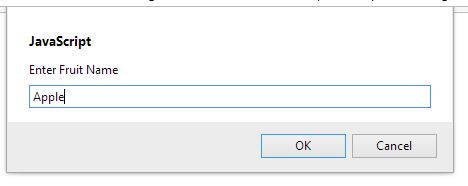
<h2>This is an example of JavaScript</h2>

<script>

</script>

</body>

</html>



<html>

<head>

<script>

ab=window.open();

ab.document.write("<font color=blue>This is written in new window</font><br>You are in new window");

</script>

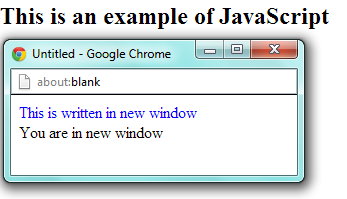
</head>

<body>

<h2>This is an example of JavaScript</h2>

</body>

</html>



**A simple form validation**

<html>

<head>

<title>First Javascript Program</title>

<script>

function frmsubmit()

{

var name=document.getElementById("uname").value;

var email=document.getElementById("email").value;

var telno=document.getElementById("telno").value;

if(name==""||name==null||email==""||email==null||telno==""||telno==null)

{

document.getElementById("msg").innerHTML="<font color=red>Some field missing</font>";

}

else

{

document.getElementById("msg").innerHTML="";

var a=email.lastIndexOf('@');

var b=email.lastIndexOf('.');

if(a<=0||(b-a)<=1)

{

document.getElementById("msg").innerHTML="<font color=red>Please enter valid email address</font>";

}

else if(isNaN(telno)==true)

{

document.getElementById("msg").innerHTML="<font color=red>Please enter valid number</font>";

}

else

{

document.frm1.submit();

}

}

}

</script>

</head>

<body>

<form name=frm1 action="" method="post">

<table>

<tr><td>Name</td>

<td><input type="text" id="uname"></td></tr>

<tr><td>Email Address</td>

<td><input type="text" id="email"></td></tr>

<tr><td>Telephone No</td><td>

<input type="text" id="telno"></td></tr>

<tr><td colspan=2><div id="msg"></div></td></tr>

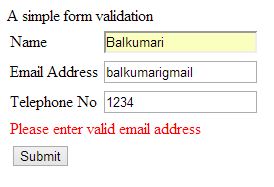
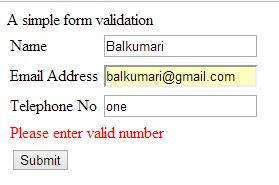
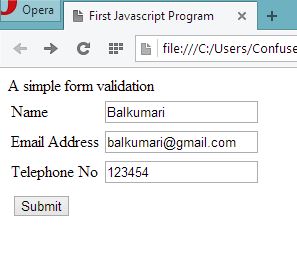
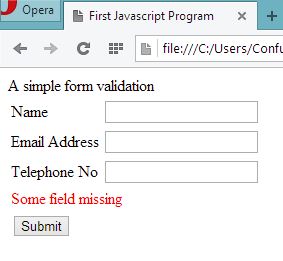
<tr><td><input type="button" value="Submit" onclick="frmsubmit();"></td></tr>

</table>

</form>

</body>

</html>



<html>

<head>

<script>

function check()

{

var pr=document.getElementById("p").value;

var rate=document.getElementById("ir").value;

var time=document.getElementById("time").value;

var total=parseInt(pr)\*parseInt(rate)\*parseInt(time)/100;

document.getElementById("ans").innerHTML="Calculated value is : RS "+total

}

</script></head><body>

<h2>This is an example of JavaScript</h2>

<form name="frm1">

<table><tr>

<td>Enter Principal</td>

<td><input type="text" id="p"></td>

</tr><tr>

<tr>

<td>Enter Rate of Interest</td>

<td><input type="text" id="ir"></td>

</tr>

<tr>

<td>Time</td>

<td><input type="text" id="time"></td>

</tr>

<tr>

<td colspan=2><input type=button value="Calculate" onclick="check();"></td></tr>

<tr><td colspan=2><div id="ans"></div></td></tr>

</table>

</form>

</body>

</html>

