# Introduction to JavaScript

JavaScript's a great language. We love it. It may not have the speed or power of Java or C, or the tight integration of ActiveX, but on the plus side it's interpreted (no messing about with compilers), fast to load, easy to debug and it embeds itself neatly into your web page.

Most important of all, it's easy to learn. Hopefully if you're fairly new to the language, these tips will help get you started. Even seasoned JavaScript coders may learn a thing or two from these tips - you never know!

This tip shows you how to get started with JavaScript. We'll look at how to embed JavaScript in your web page, and how JavaScript talks to your web browser. We'll illustrate these points by building a simple program to display an alert box when the page is first loaded.

## Where in my web page do I put JavaScript?

The simple answer is - anywhere you like! Though most folks like to put it near the top of the page in one chunk, usually below the **<body>** tag or between the **<head></head>** section, for readability and ease of debugging. Putting it in the <head></head> section has the advantage that all of the code will be loaded before the page, which is more secure.

Regardless of where you place it, you **must** enclose it within the following HTML tags:

```
<script language="JavaScript">

<!--




(place your code here)




// --></script>
```

Why? Well, **<script language="JavaScript">** tells the browser to interpret what follows as JavaScript, while the comment markers ( **<!-- // -->** ) make the JavaScript invisible to older browsers that don't support the language.

## How does JavaScript make the browser do stuff?

JavaScript talks to your browser through **objects** and **methods**. An object is something like a window, a frame, or an image - the things that make up your browser and the web pages it displays. A method is something that an object can do. For example, a window can be opened or closed.

As an example, we're going to make your browser display an alert box when your page loads. Click here to open the example page and see the alert box. Here's the function that made that extremely witty box appear:

```
function showAlert ( )

    {

        window.alert ( "ALERT! ALERT! KLINGONS

        ON THE STARBOARD BOW!!" );
```

```
}
```

See? It's easy! We're calling the **alert** method of the **window** object, to make the alert box appear. The method takes one **argument** - the text to display in the box. Couldn't be simpler.

But how does it come up automatically, when the page is first viewed? That's done with a little trick called the **onLoad** event handler. In the standard **<body>** tag at the top of the page, we add this code:

```
<body bgcolor="#FFFFFF" onLoad="showAlert()">
```

This tells the browser to execute the JavaScript function **showAlert()** when the page is first loaded. (It will also be executed when you click Reload/Refresh.)

Congratulations - if you made it this far, you now understand the building blocks of JavaScript! Now have a browse through the other tutorials in this section. Start writing your own code. There are lots of helpful books available on the subject, and some great websites too! Good luck.

## Events and Event Handlers

In this tutorial we'll introduce JavaScript's system of handling events, and describe some commonly used event handlers and the neat tricks you can do with them.

### What Are Events?

Events allow you to write JavaScript code that reacts to certain situations. Examples of events include:

- The user clicking the mouse button
- The Web page loading
- A form field being changed

### Event Handlers

To allow you to run your bits of code when these events occur, JavaScript provides us with **event handlers**. All the event handlers in JavaScript start with the word **on**, and each event handler deals with a certain type of event. Here's a list of all the event handlers in JavaScript, along with the objects they apply to and the events that trigger them:

| Event Handler | Applies To: | Triggered When: |
|---|---|---|
| onAbort | Image | The loading of the image is cancelled. |
| onBlur | Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, TextArea, Window | The object in question loses focus (e.g. by clicking outside it or pressing the TAB key). |
| onChange | FileUpload, Select, Text, TextArea | The data in the form element is changed by the user. |
| onClick | Button, Document, Checkbox, Link, Radio, Reset, Submit | The object is clicked on. |
| onDblClick | Document, Link | The object is double-clicked on. |
| onDragDrop | Window | An icon is dragged and dropped into the browser. |
| onError | Image, Window | A JavaScript error occurs. |
| onFocus | Button, Checkbox, FileUpload, Layer, Password, Radio, Reset, Select, Submit, Text, TextArea | The object in question gains focus (e.g. by clicking on it or pressing the TAB key). |

| | | |
|---|---|---|
| | Window | |
| onKeyDown | Document, Image, Link, TextArea | The user presses a key. |
| onKeyPress | Document, Image, Link, TextArea | The user presses or holds down a key. |
| onKeyUp | Document, Image, Link, TextArea | The user releases a key. |
| onLoad | Image, Window | The whole page has finished loading. |
| onMouseDown | Button, Document, Link | The user presses a mouse button. |
| onMouseMove | None | The user moves the mouse. |
| onMouseOut | Image (NOT NS4), Link | The user moves the mouse away from the object. |
| onMouseOver | Image (NOT NS4), Link | The user moves the mouse over the object. |
| onMouseUp | Button, Document, Link | The user releases a mouse button. |
| onMove | Window | The user moves the browser window or frame. |
| onReset | Form | The user clicks the form's Reset button. |
| onResize | Window | The user resizes the browser window or frame. |
| onSelect | Text, Textarea | The user selects text within the field. |
| onSubmit | Form | The user clicks the form's Submit button. |
| onUnload | Window | The user leaves the page. |

## Using an Event Handler

To use an event handler, you usually place the event handler name within the HTML tag of the object you want to work with, followed by **="SomeJavaScriptCode"**, where **SomeJavaScriptCode** is the JavaScript you would like to execute when the event occurs.
For example:

```
<input type="submit" name="clickme"

value="Click Me!" onClick="alert('Thank You!')">
```

## The Event Object

The **Event** object is created automatically whenever an event occurs. There are a number of properties associated with the **Event** object that can be queried to provide additional information about the event:

| Event Property | Description |
|---|---|
| event.data | Used by the **onDragDrop** event. Returns an array of URL's of dropped objects. |
| event.height | Stores the height of the window or frame containing the object connected with the event. |
| event.modifiers | Returns a string listing any modifier keys that were held down during a key or mouse event. The modifier key values are: ALT_MASK, CONTROL_MASK, SHIFT_MASK and META_MASK. |

| | |
|---|---|
| event.pageX<br>event.pageY | These properties hold the X and Y pixel coordinates of the cursor relative to the page, at the time of the event. |
| event.screenX<br>event.screenY | These properties hold the X and Y pixel coordinates of the cursor relative to the page, at the time of the event. |
| event.target | Returns a string representing the object that initiated the event. |
| event.type | Returns a string representing the type of the event (keypress, click, etc). |
| event.which | Returns a number representing the mouse button that was pressed (1=left, 2=middle, 3=right) or the ASCII code of the key that was pressed. |
| event.width | Stores the width of the window or frame containing the object connected with the event. |
| event.x<br>event.y | These properties hold the X and Y pixel coordinates of the cursor relative to the layer connected with the event or, for the **onResize** event, the width and height of the object after it was resized. (You can also use **event.layerX** and **event.layerY**, which do the same thing.) |

## Some Common Event Handlers

In this section, we'll look at a few of the more commonly used event handlers, and examine how they can be used.

### onChange

**onChange** is commonly used to validate form fields (see our tutorial on Form Validation with JavaScript) or to otherwise perform an action when a form field's value has been altered by the user. The event handler is triggered when the user changes the field then clicks outside the field or uses the TAB key (i.e. the object loses focus).

#### Example

This example code ensures that you type in both your first and your last names. It will bring up an alert box and refocus the text box field if you only type one word into the text box.

```
Please enter your name: <input type="text"

name="your_name" onChange=validateField(this)>



<script language="JavaScript">



function validateField ( fieldname )

{

   if ( ( fieldname.value ) &&

   ( ! fieldname.value.match ( " " ) ) )
```

```
    {
        alert ( "Please enter your first and last names!" );
        fieldname.focus ( );
    }
}
</script>
```

Please enter your name:

## onClick

The **onClick** handler is executed when the user clicks with the mouse on the object in question. Because you can use it on many types of objects, from buttons through to checkboxes through to links, it's a great way to create interactive Web pages based on JavaScript.

### Example

In this example, an alert box is displayed when you click on the link below.

```
<a href="#" onClick="alert('Thanks!')">Click Me!</a>
```

Click Me!

## onFocus

**onFocus** is executed whenever the specified object gains focus. This usually happens when the user clicks on the object with the mouse button, or moves to the object using the TAB key. **onFocus** can be used on most form elements.

### Example

This example text box contains the prompt **Please enter your email address** that is cleared once the text box has focus.

```
<input type="text" name="email_address"
size="40" value="Please enter your email address"
onFocus="this.value="">
```

Please enter your email address

## onKeyPress

You can use **onKeyPress** to determine when a key on the keyboard has been pressed. This is useful for allowing keyboard shortcuts in your forms and for providing interactivity and games.

### Example

This example uses the **onKeyPress** event handler for the **Window** object to determine when a key was pressed. In addition, it uses the **which** property of the **Event** object to determine the ASCII code of the key that was pressed, and then displays the pressed key in a text box. If **event.which** is undefined it uses **event.keyCode** instead (Internet Explorer uses **event.keyCode** instead of **event.which**).

```
<html>

<body onKeyPress = "show_key(event.which)">

<form method="post" name="my_form">

The key you pressed was:

<input type="text" name="key_display" size="2">

</form>

<script language="JavaScript">

function show_key ( the_key )

{

   if ( ! the_key )

   {

      the_key = event.keyCode;

   }
```

```
    document.my_form.key_display.value

    = String.fromCharCode ( the_key );

}

</script>

</body>

</html>
```

Click here to try it out in a new window!

## onLoad

The **onLoad** event handler is triggered when the page has finished loading. Common uses of **onLoad** include the dreaded pop-up advertisement windows, and to start other actions such as animations or timers once the whole page has finished loading.

### Example

This simple example displays an alert box when the page has finished loading:

```
<html>

<body onLoad = "alert('Thanks for visiting my page!')">

My Page

</body>

</html>
```

Click here to try it out in a new window!

## onMouseOut, onMouseOver

The classic use of these two event handlers is for JavaScript rollover images (images, such as buttons, that change when you move your mouse over them). We have a tutorial on just this topic called Rollover Buttons with JavaScript.

### Example

Here's a simple example that alters the value of a text box depending on whether the mouse pointer is over a link or not.

```
<form>

<input type="text" name="status" value="Not Over the Link">

<br>

<a href="" onMouseOver="status.value='Over the Link'"
onMouseOut="status.value='Not Over the Link'">Move
the Mouse Over Me!</a>

</form>
```

Not Over the

Move the Mouse Over Me!

## onSubmit

The **onSubmit** event handler, which works only with the **Form** object, is commonly used to validate the form before it's sent to the server. In fact we have a whole tutorial on this topic, called Form Validation with JavaScript.

### Example

This example asks you to confirm whether you want to submit the form or not when you click on the button. It returns **true** to the event handler if the form is to be submiited, and **false** if the submission is to be cancelled.

```
<form onSubmit="return confirm('Are You Sure?')">
```

```
<input type="submit" name="submit" value="Submit">


</form>
```

Submit

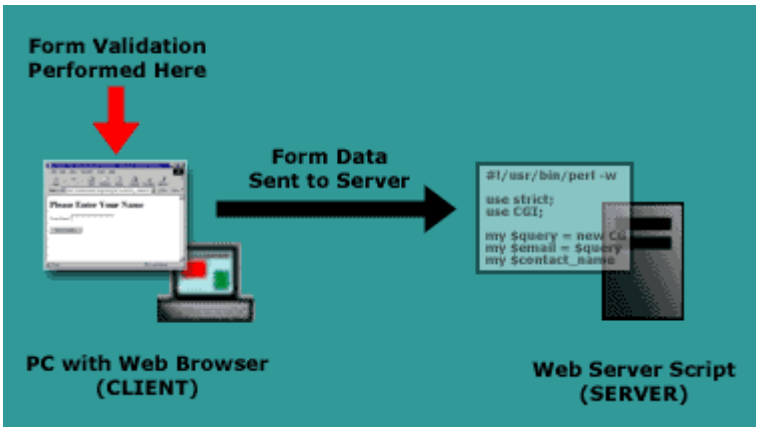
## Form Validation with JavaScript

This tutorial will show you how to create a JavaScript-enabled form that checks whether a user has filled in the form correctly before it's sent to the server. This is called **form validation**. First we'll explain why form validation is a useful thing, and then build up a simple example form, explaining things as we go along. At the end, there's a little exercise to keep you busy too!

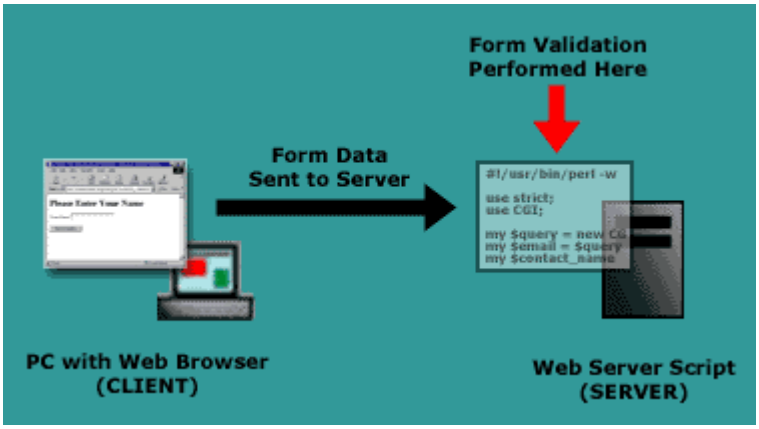
### What is form validation?

Form validation is the process of checking that a form has been filled in correctly before it is processed. For example, if your form has a box for the user to type their email address, you might want your form handler to check that they've filled in their address before you deal with the rest of the form.

There are two main methods for validating forms: **server-side** (using CGI scripts, ASP, etc), and **client-side** (usually done using JavaScript). Server-side validation is more secure but often more tricky to code, whereas client-side (JavaScript) validation is easier to do and quicker too (the browser doesn't have to connect to the server to validate the form, so the user finds out instantly if they've missed out that required field!).




**Client-side form validation (usually with JavaScript embedded in the Web page)**




**Server-side form validation (usually performed by a CGI or ASP script)**

In this tutorial we'll build a simple form with client-side JavaScript validation. You can then adapt this form to your own requirements.

## A simple form with validation

Let's build a simple form with a validation script. The form will include one text field called **Your Name**, and a **submit** button. Our validation script will ensure that the user enters their name before the form is sent to the server.

Click Here to view the form in action. Try pressing the **Send Details** button without filling anything in the **Your Name** field.

Click Here to open the source for this form in a separate window, so that you can refer to it throughout the tutorial.

You can see that the page consists of a JavaScript function called **validate_form ( )** that performs the form validation, followed by the form itself. Let's look at the form first.

### The form

The first part of the form is the **form** tag:

```
<form name="contact_form" method="post"

action="http://www.elated.com/cgi-bin/

   tutorials/javascript/contact_simple.cgi"

onSubmit="return validate_form ( );">
```

The form is given a **name** of **"contact_form"**. This is so that we can reference the form by name from our JavaScript validation function.

The form uses the **post** method to send the data off to a dummy CGI script on ELATED.com's server that thanks the user. In reality, you would of course send the data to your own CGI script, ASP page, etc (e.g. a form mailer).

Finally, the **form** tag includes an **onSubmit** attribute to call our JavaScript validation function, **validate_form ( )**, when the **Send Details** button is pressed. The **return** allows us to return the value **true** or **false** from our function to the browser, where **true** means "carry on and send the form to the server", and **false** means "don't send the form". This means that we can prevent the form from being sent if the user hasn't filled it in properly.

The rest of the form prompts the user to enter their name into a form field called **contact name**, and adds a **Send Details** submit button:

```
<h1>Please Enter Your Name</h1>
```

```
<p>Your Name: <input type="text" name="contact_name"></p>

<p><input type="submit" name="send" value="Send Details"></p>



</form>
```

Now let's take a look at the JavaScript form validation function that does the actual work of checking our form.

## The validate_form ( ) function

The form validation function, **validate_form ( )**, is embedded in the **head** tag near the top of the page:

```
<script language="JavaScript">



<!--



function validate_form ( )

{

   valid = true;



   if ( document.contact_form.contact_name.value == "" )

   {

      alert ( "Please fill in the 'Your Name' box." );

      valid = false;

   }



   return valid;

}
```

```
//-->


</script>
```

The first line (**<script language="JavaScript">**) tells the browser that we are writing some JavaScript, and the HTML comment (**<!--**) in the second line hides the script from older browsers that don't understand JavaScript.

Next we start our **validate_form ( )** function, then set a variable called **valid** to the value **true**:

```
function validate_form ( )

{

    valid = true;
```

We use this **valid** variable to keep track of whether our form has been filled out correctly. If one of our checks fails, we'll set **valid** to **false** so that the form won't be sent.

The next 5 lines check the value of our **contact_name** field to make sure it has been filled in:

```
    if ( document.contact_form.contact_name.value == "" )

    {

        alert ( "Please fill in the 'Your Name' box." );

        valid = false;

    }
```

If the field is empty, the user is warned with an **alert** box, and the variable **valid** is set to **false**.
Next, we return the value of our **valid** variable to the **onSubmit** attribute (described above). If the value is **true** then the form will be sent to the server; if it's **false** then the form will not be sent:

```
    return valid;
```

Finally, we finish our **validate_form ( )** function with a closing brace, and end our HTML comment and **script** tag:

```
}




//-->
```

```
</script>
```

That's all there is to simple JavaScript form validation! Our example is very simple as it only checks one field. Let's expand this example with a more complex function that checks lots of form fields. We'll also look at how to check other types of fields, such as checkboxes, radio buttons and drop-down lists.

## A more complex form

Let's look at a more complex validated form with some different types of form fields.

Click Here to view the form in action. Try pressing the **Send Details** button without filling in the form and see what happens.

Click Here to open the source for this form in a separate window, so that you can refer to it as we talk you through.

Like our previous example, this page has a form called **contact_form** and a function called **validate_form**. In addition to the previous text field, the form has radio buttons, a drop-down list and a checkbox.

The **validate_form ( )** function now has 3 extra checks, one for each of our new fields.

### Validating radio buttons

After the **contact_name** text box has been checked, the **gender** radio buttons are validated:

```
if ( ( document.contact_form.gender[0].checked == false )

&& ( document.contact_form.gender[1].checked == false ) )

{

   alert ( "Please choose your Gender: Male or Female" );

   valid = false;

}
```

This code checks to see whether either of the radio buttons (**Male** or **Female**) have been clicked. If neither have been clicked (**checked == false**), the user is alerted and **valid** is set to **false**.

### Validating drop-down lists

Next the **Age** drop-down list is checked to see if the user has selected an option. In the form, we named the first option in the drop-down list "Please Select an Option. Our JavaScript can then check which option was selected when the user submitted the form. If the first option is selected, we know the user has not selected a "real" option and can alert them:

```
if ( document.contact_form.age.selectedIndex == 0 )

{

   alert ( "Please select your Age." );
```

```
    valid = false;

}
```

Note that the values for **selectedIndex** start at zero (for the first option).

### Validating checkboxes

Finally, the **Terms and Conditions** checkbox is validated. We want to the user to agree to our imaginary Terms and Conditions before they send the form, so we'll check to make sure they've clicked the checkbox:

```
if ( document.contact_form.terms.checked == false )

{

    alert ( "Please check the Terms & Conditions box." );

    valid = false;

}
```

Because we set our **valid** variable to **false** in any one of the above cases, if one or more of our checks fail, the form will not be sent to the server. If the user has not completed more than one field, then they will see an alert box appear for each field that is missing.

Now you know how to write a form validation script that can handle multiple form fields, including text boxes, radio buttons, drop-down lists and check boxes!

One point to note about JavaScript validation is that it can always be circumvented by the user disabling JavaScript in their browser, so for secure validation you'll need to write your validating code in your server-side scripts. However, for day-to-day use JavaScript is a quick and easy way to check over your forms before they're sent to your server.

### An Exercise: One Field at a Time Validation

Our example script works by validating all the form fields at once. This can be a bit confusing for the user, especially if they've missed out more than one field, as they will get lots of alert boxes appearing and they might forget which fields they need to fill in!

As an exercise, try modifying the script to only prompt the user one field at a time. For example, if they miss out the **Name** and **Gender** fields and press **Send Details**, it will only prompt them for the **Name** field initially. Then, after they fill in the **Name** field and press **Send Details** again, it will prompt them for the **Gender** field.

As a finishing touch, try making the script move the cursor to the field that needs filling in each time (**Hint:** use the **focus()** method to do this).

## Opening Windows with JavaScript

One of the most useful (and quite possibly the most abused) features of JavaScript is its ability to manipulate browser windows. It can be very handy for creating a pop-up navigation window, or for making snappy websites with no menus or button bars (see one of our pagekits, **filmstar**, for an example of the latter).

Multiple pop-up windows can be a real pain, especially now that certain free web space companies are getting in on the act as a method of advertising, so go easy on them. A good rule of thumb is: if you're opening two new windows, you're opening one too many!

## What does a pop-up window look like?

Click here...

## Splendid! How do I do one?

It's really simple. Here's the function that made that window appear:

```
function openWinBoo()
    {
        NewWindow=window.open('boo.html', 'boo',
        'width=180,height=50');
    }
```

This creates a new window called "boo", which displays the HTML page "boo.html", and is 180 pixels wide and 50 pixels high.
The function **openWinBoo()** is called when you click on the link above. The code for the link looks like this:

```
Click <a href="javascript:openWinBoo()">here</a>
```

The **javascript:** bit tells the browser to call a JavaScript function - in this case, our **openWinBoo()** function.

Let's take a closer look at our function. The last argument between the parentheses specifies how our new window will look. In our example, we've just specified the width and height with '**width=180**,**height=50**'. However, there are many properties of the window that are under our control.

| Property | Meaning |
|---|---|
| **directories** | The "what's new/what's cool" bar (Netscape only!) |
| **location** | The box allowing the user to type a URL |
| **menubar** | The menu bar (File, Edit, etc.) |
| **resizable** | The user can resize the new window by dragging |
| **scrollbars** | The new window has scrollbars |
| **toolbar** | The toolbar (Back, Forward, etc.) |

Each of these properties can have the value of **yes** (the feature will appear in the new window) or **no** (the new window will have that feature disabled).

An important point to note about these properties is that they mustn't have spaces between them. For example, **'width=180,height=50'** will work, but **'width=80, height=50'** won't.
For example, this function creates a new window with just the menu and the URL entry box:

```
function openWinMenuUrl()
    {
        NewWindow=window.open('menu_url.html', 'menu_url',
        'width=400,height=100,location=yes,menubar=yes');
```

```
}
```

Click here to see it in action.

Now you know how to make a new window appear, and how to make that window look the way you want it. Try playing with different properties in the **window.open** method and enjoy the power of JavaScript!

## Rollover Buttons with JavaScript

These days every man and his dog is using "rollover", or "mouse-sensitive" buttons on their site. Usually these are achieved through JavaScript, although it's possible to do the same thing in DHTML, Java or (perish the thought) ActiveX. JavaScript is the simplest one to understand, though. If you don't know what I'm on about, move your mouse over this baby:

Get the idea? **Note**: You need to be using Netscape 3 or greater, or IE 4 or greater, to see rollovers, as only these newer browsers have the ability to change images in JavaScript.

### Where to use 'em

Rollover images are great whenever you want to make it obvious that the user should click on them (for example, buttons and menus).

### Where not to use 'em

In the bath.

### OK wise guy - how's it done?

Well it all hangs around JavaScript's **Image** class. Let's look at how we did the rollover button above, to help us understand this class. Click here to view the HTML source of this page in a separate window, so you can follow through the JavaScript it contains.

The first thing we need to do is create two images - one for the "normal" button, and one for the "active" (rollover) button. We do this with the **new Image** constructor, which simply takes two values - the image **width** and **height** - and makes the image object:

```
eg_on = new Image ( 33, 33 );;

eg_off = new Image ( 33, 33 );
```

We've chosen the image files **images/eg_on.gif** and **images/eg_off.gif**. In this example, the file names are the same as the image object names, but they don't have to be! It's just easier to follow, that's all.

### Changing the images

Now, our example on this page only has one rollover button; however, often you'll want several on one page, as in a menu. To make this easier, we've created a couple of **functions** to handle the actual image changing. That way, you can just call the function for each of your buttons, without having to duplicate lots of code.

Here are those functions, called **button_on** and **button_off**:

```
function button_on ( imgName )

    {
```

```
        if ( version == "n3" || version == "e4" )
        {
            butOn = eval ( imgName + "_on.src" );
            document [imgName].src = butOn;
        }
    }

function button_off ( imgName )
    {
        if ( version == "n3" || version == "e4" )
        {
            butOff = eval ( imgName + "_off.src" );
            document [imgName].src = butOff;
        }
    }
```

Note that these functions rely on your **Image** names ending in **_off** (for the normal image), or **_on** (for the rolled-over image). That's just our convention; change it to something else if you like, but remember to change it both in the two functions above, and in your **new Image** statements at the top of the script!

The two functions are nearly identical. Both take the **Image** object specified by **imgName**, which is the name we give the button in the HTML (in this case **eg**), and replace its source with either the **_off** or the **_on** image's source.

### I'm confused! What are all these different images?

There are three **Image** objects at work here:

**eg** The actual image in the HTML page (see below); initially it points to **images/eg_off.gif**

**eg_off** The "off" version of the image (which also points to **images/eg_off.gif**)

**eg_on** The "on", or rolled-over, version of the image, which points to **images/eg_on.gif**

The functions **button_off** and **button_on** simply replace the image pointed to by **eg** with the images pointed to by **eg_off** and **eg_on** respectively. Easy-peasy!

### Browser sniffing

Note also the use of the **version** variable - this is defined right at the start of our script, and is used to determine which browser we are running. This is often referred to as sniffer code:

```
if ( browserName == "Netscape" && browserVer >= 3 )

   version = "n3";

if ( browserName == "Microsoft Internet Explorer" && browserVer >=4 )

   version = "e4";
```

### The HTML bit

The last piece of the puzzle is to make the call to our JavaScript functions from within the HTML itself. To do this, we need to have an active link (an **<a href>** tag) around our rollover button. As our button doesn't need to link anywhere, we link it to # (an empty anchor).

```
<a href="#"

onmouseout="button_off('eg'); return true"

onmouseover="button_on('eg'); return true"><img

src="images/eg_off.gif" border="0" width="33"

height="33" name="eg"></a>
```

The two attributes **onmouseout** and **onmouseover** tell the browser to execute the JavaScript within the quotes (" ") when the mouse moves out of or over the button respectively. Within the quotes, we make a call to the appropriate image-changing function (**button_off** or **button_on**) that we defined above.

Last - but by no means least - the button is given a name using the attribute **name="eg"**. This name is very important, as it is used by the **button_on** and **button_off** functions to determine which image to change. Putting the name in the HTML implicitly tells the browser to create a new **Image** object, in this case called **eg**. When using many rollovers in one page, make sure you use a different name for each button!

### Go for it, my son!

Now you should be able to take the above code and use it to make your own rollover images. You can make as many rollovers as you like in one page - just remember to give each one a unique **name**, and to call the functions **button_off** and **button_on** with that name.

## JavaScript and Forms

The **FORM** tag is usually used to create forms for sending to CGI scripts or other server-based software, but there's a lot that can be done at the browser end as well - with JavaScript! You can validate the data entered by the user before it's sent to the CGI script (this saves time and bandwidth); you can get the user to talk to your JavaScript program through form fields; or you can even design a menu system using forms and JavaScript. In fact, we're going to show you how to do just that.

### A menu using JavaScript and forms

To see this in action, take a look at a page from one of our PageKits, filmstar. In the top left of the window you'll see a drop-down menu. When you pick a page from the menu, it'll take you straight to that page. Cool, huh? Now this is how we did it...

First, we make an **array** containing the URL's of all the pages in our drop-down menu. An array is just a list of objects - in this case, URL's. To make this easier we write a function to build the array:

```
function buildArray()

{

    var a = buildArray.arguments;

    for (i=0; i<a.length; i++)

    {

        this[i] = a[i];

    }

    this.length = a.length;

}
```

Then we call that function, passing it the URL's we want in the array:

```
var urls1 = new buildArray("",

"about.html",

"new.html",

"portfolio.html",

"guests.html",

"links.html",

"mailto:you@wherever.whatever?subject=the site");
```

We're naming our array **urls1**, so that if we wanted more than one drop-down menu in the page, we could easily add more (**urls2**, **urls3** etc). This is also catered for in the next function, **go**, which loads the new pages into the window when the user selects them from the menu:

## Go, go, go! Let's see some action...

```
function go (which, num, win)

{
```

```
        n = which.selectedIndex;

        if (n != 0)

        {

                var url = eval("urls" + num + "[n]")

                if (win)

                {

                        openWindow(url);

                } else

                {

                        location.href = url;

                }

        }

}
```

The **go** function takes 3 arguments. The first one, **which**, is the object whose event handler called the function - in this case, the drop down menu. This is passed so that the function knows where to find the menu object! The second argument, **num**, refers to the array of URL's we mentioned above; in this case there's only one array, **urls1**, so we'll pass the value **1** to the function. The third argument, **win**, is a boolean value (either true or false) - **true** means that a new window should be created for the selected page, while **false** means that the selected page should be opened in the current window.

Now the function itself should be fairly self-explanatory. The first line, **n=which.selectedIndex**, gets the value of the **selectedIndex** property from the object **which** (our drop down menu) - in other words, **n** now holds the position of the selected item in the menu (a number between 1 and 7). **n** is then used to find the URL to display, by looking it up in our **urls1** array. Finally the URL is displayed, either in a new window (if **win** was true) or in the same window.

## The HTML bit

The last step is to call the **go** function from the drop-down menu itself. Here is the HTML for the menu:

```
<form name="form1">



<select name="menu1" onChange="go(this, 1, false)">

<option>f i l m s t a r
```

```
<option>about

<option>what's new?

<option>portfolio

<option>guests

<option>links

<option>m a i l  u s

</select>



</form>
```

The **select** tag generates a drop-down menu. As you can see, this tag has an event handler **onChange**, which is called when the user picks an item from the menu. We use this handler to call our function **go** with the reference to the drop-down menu object (**this**), the URL array we're using (there's only one, so **1**) and whether we want the links to open in a new window (**false**, or no)

That's all there is to it! You can take the above code and change the menu text and URL's to make menus for your own sites. It's a great way to make a menu if screen real-estate is at a premium. Plus it teaches you a bit about using JavaScript with forms!

*(we can't remember from which source we found this tutorial – it is definitely not made by us)*