

Javascript for beginners

© Copyright Martin Baier

Translated from the German by Linda L. Gaus

© Copyright 2000 Author and KnowWare

Acrobat Reader: How to ...

F5/F6 open/closes bookmarks - **F4** open/closes thumbnails

In menu View you can set, how the file is displayed

CTRL+0 = Fit in Window, **CTRL+1** = Actual size, **CTRL+2** = Fit width

You can set **SINGLE PAGE**, **CONTINUOUS VIEW** or **CONTINUOUS FACING**

.. try them out and you will see the differences.

Navigation

ARROW LEFT/RIGHT: forward/backwards one page

ALT+ARROW LEFT/RIGHT: same as in a browser: forward/back

CTRL++ zooms in **AND CTRL +-** zooms out

www.knowwareglobal.com

The Basics.....	5	Strings	43
The Necessary Software	5	The String Object	44
HTML	5	length	44
What are HTML Pages?	5	substring	44
Brief HTML Reference Guide	5	toLowerCase	44
HTML and JavaScript.....	7	toUpperCase.....	44
Incorporation in the Header.....	7	Moving Text	44
Carrying out Code Given Particular Actions.....	8	User-Defined Objects.....	46
Incorporation in the Body	8	Arrays.....	46
First JavaScript Programming	9	Working with Frames.....	46
Hello World	9	Quiz	49
Hello World without Parameters	9	The Explorer.....	58
Hello World with Parameters	10	The Project.....	58
What time is it?	11	The Practice	58
Page Reference	12	The Main Page	58
Event Handler.....	13	The Content Page.....	60
onLoad	13	The Explorer Page	60
onUnload.....	13	Customization.....	65
onMouseOver	14	Reserved Words	66
onMouseOut	14	The Last Word...	66
onFocus.....	14		
onBlur	16		
onChange	16		
onClick.....	17		
javascript.....	18		
onSubmit.....	18		
Functions.....	19		
Variables	20		
Local Variables	20		
Global Variables	21		
Mathematical Operations	22		
Repeated Performance.....	23		
Looping with for	23		
Looping with while.....	25		
Conditional Operations.....	27		
Standard Objects.....	29		
document.....	29		
Colors in the Document.....	29		
Document Properties	31		
Pictures in a Document.....	32		
document.frames	35		
document.forms	38		
Text Entry Fields	38		
Radio and Check buttons.....	38		
Drop-Down Lists.....	38		
Pizza Service	39		
Euro Calculator	41		

Introduction

Everyone who is the least bit familiar with the Internet eventually wants to represent him or herself there with a home page. But the common page-building programs like Netscape Composer or Microsoft Frontpage no longer suffice for creating anything more than a very mediocre home page. Anyone who wants to have a really cool home page must know a little more than the countless amateurs who are out there on the Internet. The easiest and best tool for creating a truly attractive and interactive home page is called JavaScript.

The beautiful thing about JavaScript is that the knowledge and system-related prerequisites for learning the language are relatively low. You just need to know some HTML. And I'll teach you the most crucial things in the first section of this booklet. If you'd like to delve deeper, I recommend the KnowWare booklet "Homepages for Beginners" by Johann-Christian Hanke. Since JavaScript is platform-independent, it can be used on almost any Mac or PC. As far as software is concerned, you'll need just an Internet browser (preferably Netscape Navigator or Microsoft Internet Explorer) and a simple ASCII text editor, for example, the one that comes with Windows. ASCII text consists of unformatted letters, that is, for every letter, you need to have one byte of hard disk space available. By contrast, Microsoft Word formats text with fonts, colors, etc. and it's not really suited to serve as an ASCII text editor.

Finally, here's a hint for reading this booklet: everything typed in **Courier** is code. To test it on the computer, you'll have to type it in. You'll find current information about JavaScript and updates to the contents of this booklet on its companion page.

The address is
www.knowwareglobal.com/javascript

Go have a look!

I'd like to thank Dipl.-Informatiker Reinhold Baier for proofreading this booklet – thanks to him, you've been spared many content- and language-related mistakes.

Finally, I'd like to ask one thing of you: if you have comments about this booklet, whether positive or negative, e-mail them to me! I'm always very open to having you do that. I wish you much enjoyment in programming and I'm sure that your home page will, in the future, set itself apart from the masses of the WYSIWYG editor sites because of JavaScript.

Weilheim, July 2000

Martin Baier (martin.baier@gmx.net)

The Basics

This section will give you an overview of HTML and inform you about how to incorporate JavaScript into HTML pages. If you're already familiar with HTML and you've already worked with JavaScript, you can skip right over this section.

The Necessary Software

As I've already mentioned, all you'll need is a text editor and a browser. Text editors come with practically every operating system. Under Windows 95/98, the editor can be found by clicking [Start/ Programs / Accessories / Notepad](#). You'll find browsers that you can download for free over the Internet at the Netscape Website (www.netscape.com) and the Microsoft Website (www.microsoft.com). Given an average downloading speed and average online costs, however, it's cheaper to buy a computer magazine that contains a CD. Many of these "silver disks" contain the latest versions of the browsers.

HTML

What are HTML Pages?

HTML is a text-layout language, with help of which the most diverse systems can produce nearly identical results. This is due to the fact that the files in which the HTML code is saved, that is, files with the endings `*.HTM` or `*.HTML`, contain only ASCII text. The code in these files specifies, for example, which background color, which text color, which text and pictures in which order the page should contain. In order to make this topic more concrete, here's a brief introduction to HTML.

Brief HTML Reference Guide

In a nutshell, HTML consists of so-called tags, which are always placed inside pointy brackets `<>`. These tags are, in turn, divided into those that cause a certain action (a line break, for example) and those that format the text (italics would be an example of this). The text formatting tags require a companion tag to the introductory tag at the end of the text you'd like formatted a particular way. A few practical examples will help you understand the functions of these tags:

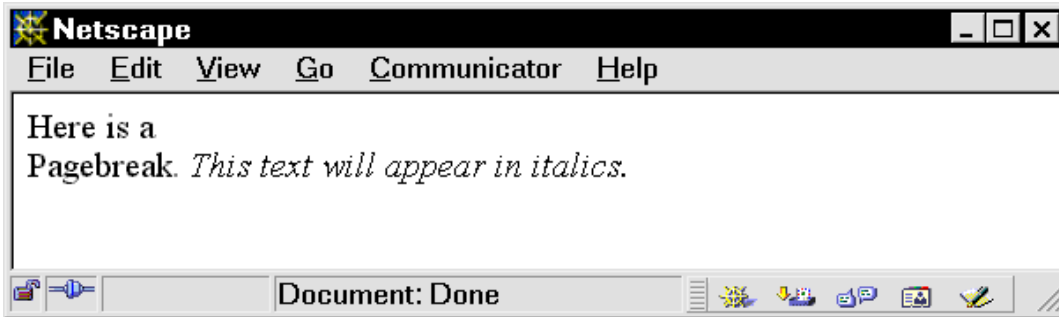
```
Here is a <br> Pagebreak.  
<i>This text will appear in italics.</i>
```

These HTML code fragments must be incorporated into the basic HTML structure. The complete code would look like this:

```
<html>  
<head>  
<title>Title of the Page (appears in the browser in the title  
line)</title>  
</head>  
<body>  
Here is a <br> Pagebreak. <i>This text will appear in italics.</i>  
</body>  
</html>
```

The entire source code must be saved in an ASCII file. But the file extension must be either `*.HTM` or `*.HTML`, not `*.TXT`. Be careful: many text editors, especially Windows editors, save files whose names are given as `*.HTM` as `*.HTM.TXT`. In this case, you must change the filename manually in the file manager or Windows Explorer.

Now you need to start the browser, either Netscape or Microsoft Internet Explorer, and open the file you've just created. Usually, you do this by selecting **File / Open**, but this can differ according to which version of which browser you're using. The page should look like this (this page was displayed with Netscape Navigator 4.7):



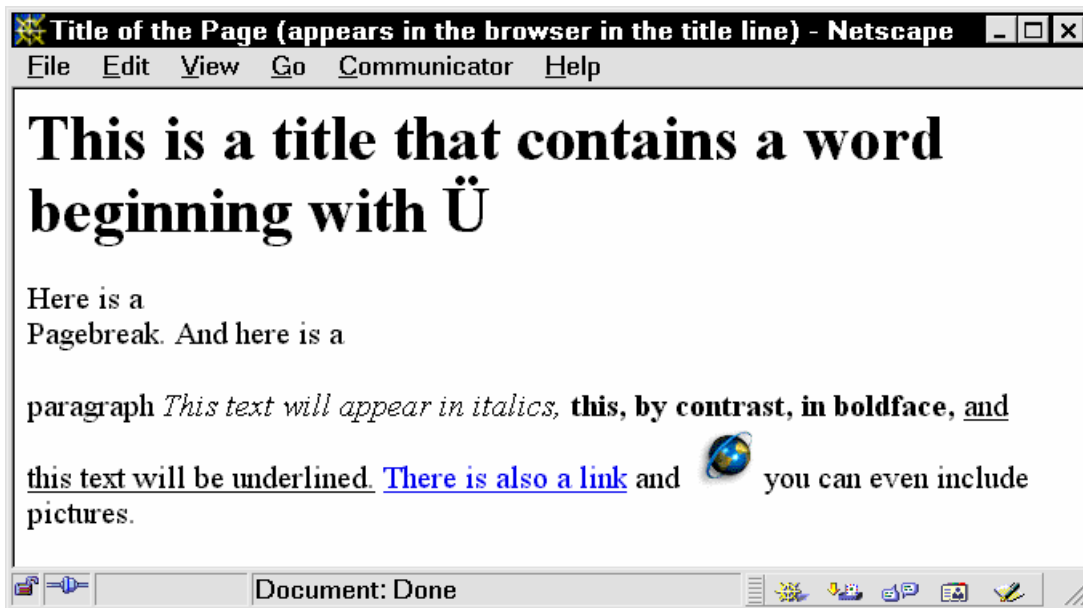
Here are a couple more things you should know – tags that begin with / always indicate the end of text formatting and `Ü` is the international way of creating a `ü`:

```
<html>
<head>
<title>Title of the Page (appears in the browser in the title
line)</title>
</head>
<body>
<h1>This is a title that contains a word beginning with &Uuml;</h1>
Here is a <br> Pagebreak. And here is a<p> paragraph
<i>This text will appear in italics,</i> <b>this, by contrast, in
boldface,</b> <u>and this text will be underlined.</u> <a
href="page2.htm">There is also a link</a> and  you can even include pictures.
</body>
</html>
```

Before you can view the page, you'll need to save a file with the name **PICTURE.GIF**. This file can contain any image you want. For example, you can create a picture using Paint under Windows 98. When saving the file, you need to make sure that it has the extension **.GIF**. Alternatively, you can go out and find a graphics file, a finished **.GIF** file, somewhere on the Internet.

If you want the link to work, you'll need to create another HTML file, which in our example, is called **PAGE2.HTM**. This name appears in the source code.

And don't forget to save your files! The result should look like this when viewed with your browser:



HTML offers you many other possibilities. But since this booklet is concerned primarily with JavaScript, I'll just refer you to some appropriate literature: The KnowWare booklets "WWW – Create Homepages Yourself" by Achim Schmidt and "Homepages for Beginners" by Johann-Christian Hanke are excellent. Naturally there are also many more comprehensive books about HTML available in bookstores. You'll find more information about free HTML documentation on the Internet at www.knowwareglobal.com/javascript/.

HTML and JavaScript

Before you begin programming, have a look at this general information about how to incorporate JavaScript into HTML pages. First off, you should know that JavaScript is a scripting language. That is, the code is not compiled (translated into machine language), but instead it appears as ASCII text in an HTML file. For integrating it into the HTML code, you have three possibilities:

Incorporation in the Header

The first possibility is to incorporate the source code into the header of the HTML file. Here you can write code that you'll access later using one of the two other possibilities for incorporation. The source code for this option looks like this:

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
Definition of Functions and Variables
//-->
</script>
...
```

The Tag `<script language="JavaScript">` introduces the JavaScript source code, the tag `</script>` ends it.

The codes `<!--` and `//-->` cause source code from older browsers, ones that don't support

JavaScript, to be hidden.

Within these tags, you can define functions and variables, but we'll get to that later.

Carrying out Code Given Particular Actions

The second possibility is to carry out JavaScript commands given a surfer's particular actions. Such actions can include the loading or leaving of a page or the following of a link with the mouse. In the following example, the function `hello` is supposed to be carried out when the page is loaded. This must be defined in advance in the header of the HTML file (see above, incorporation in the header).

```
...  
<body onLoad="hello()">  
...
```

The function could, for example, greet visitors to the site with a message on the screen.

Incorporation in the Body

In addition, you can incorporate JavaScript commands into a particular part of the page when you're building the page. This is useful if, for example, you'd like JavaScript to incorporate supplements such as text directly into the HTML file. The following example is supposed to display the text "It is xx.xx O'Clock!", replacing the letters with the actual time. The function `write_time` must be defined in the header.

```
<body>  
It is  
<script>  
write_time()  
</script>  
O'Clock!  
</body>
```

First JavaScript Programming

All of the foregoing examples may seem very abstract to you; in the following, we'll turn our attention to practical examples that will help you understand these things.

Hello World

Hello World without Parameters

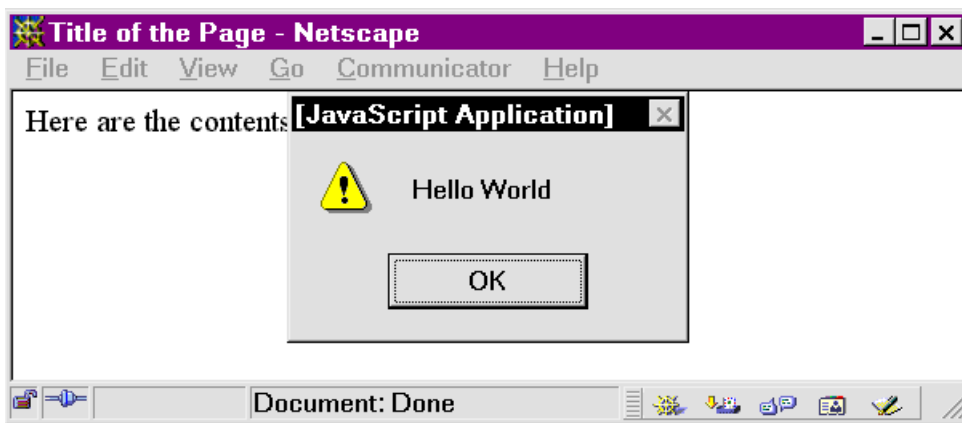
A program that is supposed to display the text "Hello World" on the screen in some way has emerged as a classic example when introducing a programming language. We want to define a function in JavaScript that will display this text as a message. To do this, we'll need the following code:

```
<html>
  <head>
    <title>Title of the Page</title>
    <script language="JavaScript">
      <!--
      function hello() {
        alert("Hello World")
      }
      //-->
    </script>
  </head>
  <body onLoad="hello()">
    Here are the contents of the page!
  </body>
</html>
```

The code up to the `<!--` should already be familiar to you.

The following line, `function hello() {`, defines a function. `function` is a reserved word that must be in this position. `hello`, by contrast, is the name of the function, which you can choose as you like. The only thing you have to watch out for is that you don't choose a reserved word as your function name – it's best to give functions meaningful names. By the way – reserved words are words that can only be used in Java Code itself, not as function or variable names. On page 65 you'll find a list of reserved words.

Further in the code: `()` means that the function isn't expecting any parameters, that is, it doesn't need any values. Our next example will demonstrate the use of functions with parameters. `{` must follow `)`. The curly bracket specifies that the content of the function begins here. `alert` is the command for displaying something in the form of a message. It takes a text parameter, which must be placed in quotation marks. `}` marks the end of the function's content. The body tag looks like this: `onLoad="hello()">`. This means that our function, again without parameters, should be called when the document is loaded. The whole thing should be saved as a normal HTML file and called up in the browser – the results look like this:



Hello World with Parameters

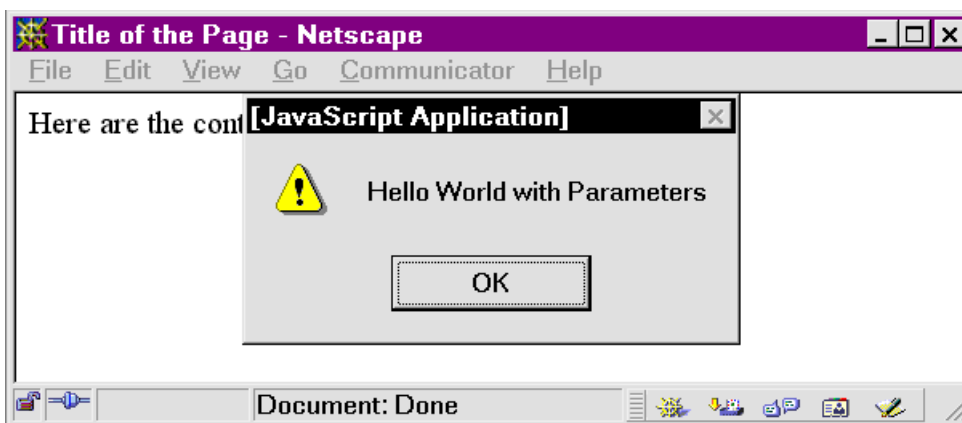
Now change the function as follows:

```
function hello(result) {  
  alert(result)  
}
```

And the body tag must be changed as follows:

```
<body onLoad="hello('Hello World with Parameters')">
```

What happens? The browser reads the body tag and encounters there the instruction to carry out the function `hello` with the text parameter `Hello World with Parameters`. The parameter is a string – and so in the source code, it must be placed in single quotes. The function `hello` is called and the string given in our example is written to the variable `result`. A variable is a space in your computer's memory that can contain numbers, text, etc. – more about this on page 20. Then the function is carried out; the browser finds the variable `result` as a parameter in the `alert` command and its value is written out as a message. In this example, using parameters doesn't make much sense – but it's important to understand how they work.



What Time is it?

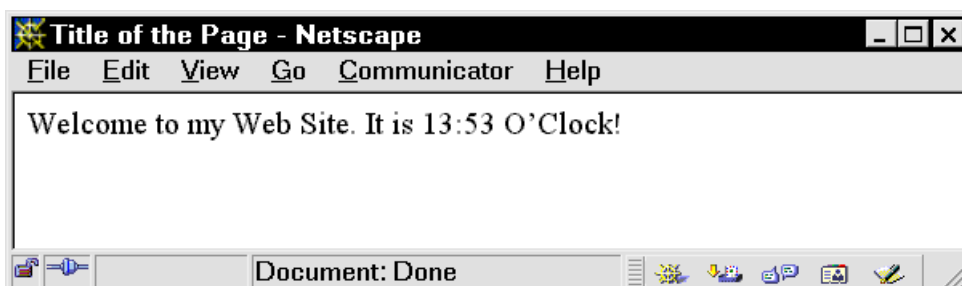
In the two foregoing examples, a function was incorporated into the header and carried out when a particular action occurred – in the examples, that action was the loading of the page. In the following example, we'll define another function, which is supposed to tell us the current time. In contrast to "Hello World," however, this function will be incorporated into the body since the current time is supposed to appear in the middle of the text on the page. This requires the following source code:

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function time(){
time=new Date()
document.write(time.getHours() + ":" + time.getMinutes())
}
//-->
</script>
</head>
<body>
Welcome to my Web Site. It is
<script>
time()
</script>
O'Clock!
</body>
</html>
```

To understand this program, several explanations are necessary:

With the first command in the function – `time=new Date ()` – all of the data concerning the current date and time are written to the object `time`. Instead of `time`, you can use any other name, but make sure that it's one that will be meaningful to you, the program's author. The time and date are now saved in `time`. Everything that's inside the parentheses after `document.write` will be written directly into the HTML document. In our case, this means the hour (`time.getHours`), a colon ("`:`") and the minutes (`time.getMinutes`). The hour and minutes are read from the object `time` using the commands issued. Then, the individual elements are connected using the plus sign (+).

The function will then be carried out within the document so that the current time will be written directly into the text. You can already see results – even such simple programs will set your Web site apart from others:



Page Reference

Before I explain the individual elements of JavaScript, here's a last practical example. At the bottom of the browser window, you'll find the status line. If you pass the mouse cursor over a link – without clicking – you'll see the file and pathname to which the link will take you. Let's try this with the following file:

```
<html>
<head>
<title>Title of the Page (appears in the Browser in the title line
)</title>
</head>
<body>
If you click here, you will go to <a href="page2.htm">Page 2</a>!
</body>
</html>
```

In the browser, you'll see the following in the status line:

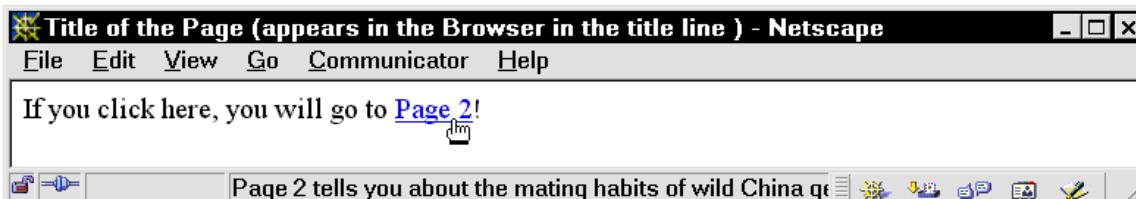
file:///C:/Files/Knowware/JavaScript/page2.htm:



If, in the reference tag, you add a JavaScript reference, the corresponding part of the file will look like this:

```
If you click here, you will go to <a href="page2.htm"
onMouseOver="window.status='Page 2 tells you about the mating habits
of wild China geese';return true" onMouseOut="window.status=' '>Page
2</a>!
```

The effect of the change is that if you pass over the link with the mouse (**onMouseOver**), you'll see a description of the target reference in the status line, which disappears again if you move the mouse away from the link (**onMouseOut**).



Event Handler

Event handler reminds you of the “Hello World” program? There you saw the following code line:

```
<body onLoad="hello()">
```

Earlier, I mentioned that the function `hello` was carried out given particular actions. To accomplish this, you needed the `onLoad` command. Commands of this kind – that is, commands that are incorporated into the HTML source code and that carry out a predefined function or command given particular actions – are called event handlers. All event handlers begin with `on...`. The `onLoad` event handler used in our example means essentially “upon loading.” Just after the event handler you’ll notice an equals sign, then the JavaScript commands to be executed in quotation marks. Now we’ll have a look at the most important event handlers.

onLoad

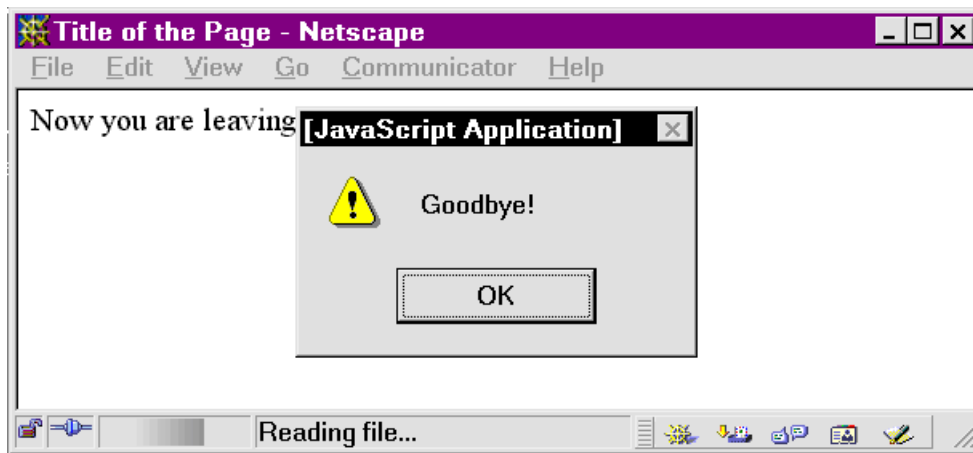
You’ve already met this event handler in the “Hello World” program in the foregoing section. It is activated when an HTML page loads.

onUnload

...is the opposite of the event handler `onLoad` and is activated when an HTML page closes. The following example is a transformation of the “Hello World” program. When leaving the page, it displays the message “Goodbye.” This could, for example, happen when a link is activated. If the link in our example is to function properly, you must create one further HTML page, called here `page2.htm`.

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function goodbye() {
alert("Goodbye!")
}
//-->
</script>
</head>
<body onUnload="goodbye()">
Now you are leaving this page <a href="page2.htm">for another</a>.
</body>
</html>
```

The result should look like this in your browser as soon as you've left the page:



onMouseOver

The event handler `onMouseOver` is probably the most-used event handler. It is used within the reference tag and becomes active if you touch the reference area with the mouse. An example of how this event handler is used is the page reference example in the previous section.

onMouseOut

`onMouseOut` is the opposite of `onMouseOver`. If you display text in the status line because you've used an `onMouseOver` event handler, you'll need to remove the text as soon as the mouse leaves the link area. The event handler `onMouseOut` was also used in the page reference example in the previous section.

onFocus

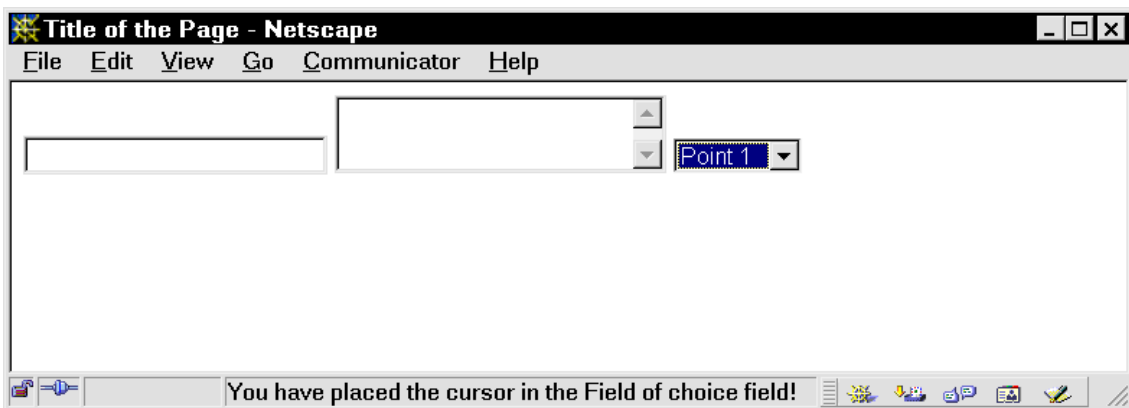
Now you'll meet a group of event handlers that are used in forms. The HTML code for forms is practically self-explanatory – if you want to know more about this topic, I would encourage you to immerse yourself in the appropriate HTML literature.

`onFocus` means “when the cursor is placed on this element.” This event handler is placed in one- and multiple-line entry fields as well as in drop-down lists, for example, in order to check zip codes that the user has entered for typos or extra digits. In our example, we assume that the surfer is dumb and tell them in the status line that they have to place the cursor on one of the three form elements. In this case, the function `message` is called up with a text parameter. This function displays the parameter that it has been given in the status line, so it functions as the page reference program does.

The code looks like this:

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function message(field) {
window.status="You have placed the cursor in the " + field + "
field!"
}
//-->
</script>
</head>
<body>
<form>
<input type="text" onFocus="message('one-line text')">
<textarea rows=2 cols=20 wrap=virtual onFocus="message('multiple-
line text ')"></textarea>
<select onFocus="message('Field of choice')">
  <option>Point 1
  <option>Point 2
  <option>Point 3
</select>
</form>
</body>
</html>
```

If the user places the cursor in the field of their choice, the following message will appear:

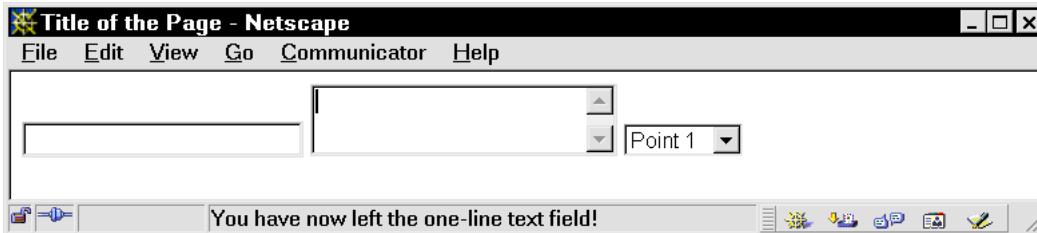


onBlur

`onBlur` is the opposite of `onFocus`. This event handler is used with exactly the same elements, namely one- and multiple-line text fields and drop-down lists. It becomes active if the focus is removed from the element, for example, as soon as the mouse is clicked on another element. Its use is parallel to that of `onFocus`. In the interest of holding source code to a minimum, we'll just show the changes to the last program:

The line `window.status="You have placed the cursor in the" + field + " field!"` must be replaced by `window.status="You have now left the " + field + " field!"`.

All three `onFocus` event handlers must be changed to `onBlur` commands. In the browser, the whole thing should now look like this:

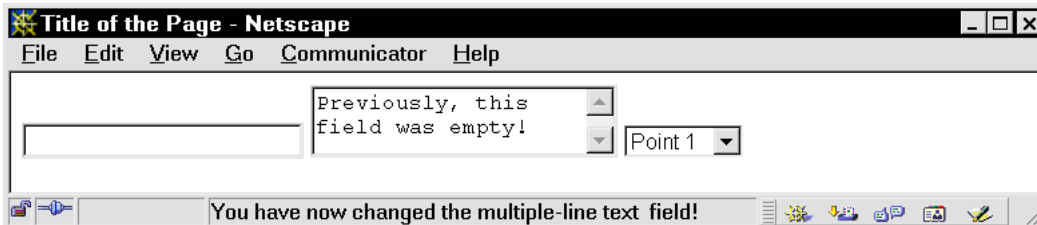


onChange

This event handler is used much like `onBlur`; it's activated as soon as the user leaves an entry field – provided that the field's value has changed. In drop-down lists, it's activated solely by the changing of the value. Because of the similarity between the `onChange` and `onBlur` event handlers, a transformation of the `onBlur` program will serve as an example:

All three `onBlur` statements must be replaced by `onChange` ones and "left" in the message must be changed to "changed."

If you save the source code and display it in your browser after making the changes and leaving the multiple-line text field, the whole thing should look like this:

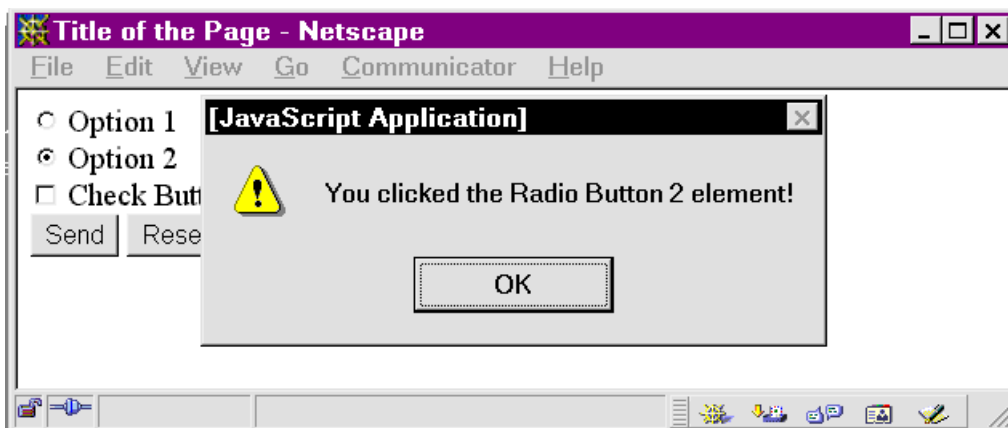


onClick

The `onClick` event handler is activated by a click on a form element. This can mean a radio or check button, but also submit, reset, or a user-defined button. In our example, if you click on a form element, a message should appear that tells you which element you clicked. Here is the source code:

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function message(element) {
alert("You clicked the " + element + " element!")
}
//-->
</script>
</head>
<body>
<form>
<input type="radio" name="Radio" onClick="message('Radio Button
1') ">Option 1<br>
<input type="radio" name="Radio" onClick="message('Radio Button
2') ">Option 2<br>
<input type="checkbox" onClick="message('Check Button') ">Check
Button<br>
<input type="submit" value="Send" onClick="message('Send Button') ">
<input type="reset" value="Reset" onClick="message('Reset Button') ">
<input type="button" value="Mine" onClick="message('My very own
Button') ">
</form>
</body>
</html>
```

The result in the browser, if you click the radio button Option 2, looks like this:



javascript

You can even use the `onClick` event handler in references, for example, like this:

```
<a href="xy.htm" onClick="message('Text')">Here is the Link!</a>
```

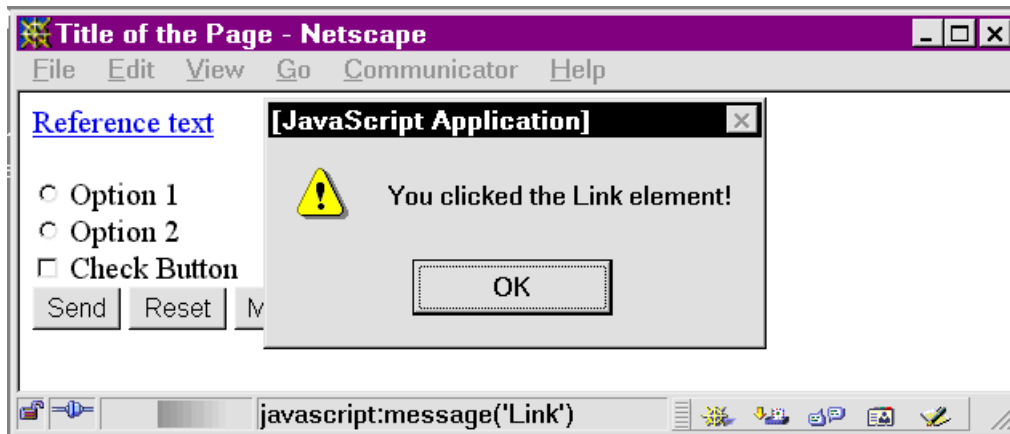
But since such links usually carry out just one JavaScript command and don't take you to another page, there is a different event handler for references, which really isn't one at all since it doesn't begin with `on...`; it begins with `javascript`.

The following exemplary source code can be incorporated into the foregoing example, just in front of

```
</body>:
```

```
<a href="javascript:message('Link')">Reference text</a>
```

If you click the reference in the browser, you'll see the following result:



onSubmit

This event handler takes care of the submission of a form. Therefore, it has the same function as `onClick` on the Send button. In our example, in the line

```
<input type="submit" value="Send" onClick="message('Send Button')">
```

you could delete the `onClick` event handler so that the line looks like this:

```
<input type="submit" value="Send">
```

But then you would have to make the following addition to the form tag:

```
<form onSubmit="message('Send Button')" >
```

The result would be the same.

Now you're familiar with the most important event handlers. There are many others that will not be described in this booklet since they are beyond its scope. But you'll find information about them on the home page for this booklet, which you'll find at www.knowware.de/javascript

Functions

An event handler carries out a function as soon as it's activated. This can be a predefined function. For example, the issuing of a message can be incorporated into the body tag:

```
<body onLoad="alert('Welcome!')">
```

The drawback to these predefined functions is that they can only each carry out one action. If you want to execute multiple JavaScript commands when an event handler is activated, you must define your own function. This happens, as we've already seen several times, in the header of the HTML file. As an example, let's look at the "Hello World" program again (this time we'll look at the whole header):

```
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function hello(result) {
alert(result)
}
//-->
</script>
</head>
```

The keyword `function` always introduces a function. After a blank space comes the function's name – in our example `hello` – which must conform to the naming conventions of JavaScript with respect to reserved words. For more information on naming conventions, see page 58. Then comes a parenthesis `(`. Now you must decide whether the function will take one or more parameters or whether it – as in our example – should use the resulting text or another value. If there are no parameters, you must still place a closing parenthesis `)` immediately next to the opening one. If, by contrast, you want to use parameters, these will be stored in variables whose names you must indicate here. When choosing variable names, you should be aware of the same issues as when choosing function names. If you're using more than one parameter, separate them with commas. In such a case, the first line of our function would look like this:

```
function hello(result,other_value) {
```

The entire contents of the function, that is, all commands that the function should carry out, must be placed in curly brackets (`{` and `}`). The function has now been defined. In our example, the function will be called as follows:

```
<body onLoad="hello('Hello World with Parameters')">
```

The function call is constructed like its definition, but instead of the variable name, the values for these variables are given – in our example `'Hello World with Parameters'`. You have to watch that the text parameters are placed in quotations. Single and double quotations are handled the same way. Expressions must also be placed in the same quotation marks. If you're nesting expressions, you must use different kinds of quotation marks. In the function call, the curly brackets disappear.

Variables

A variable is a space in your computer's memory that is reserved for a program – in our case for your JavaScript application. This place in memory can hold a number or text, but under no circumstance can a variable switch between these two types of information. One variable cannot first hold a number and then later text, or vice versa. We've already worked with variables numerous times. Here's how we used them in functions as parameters, as in the "Hello World" program:

```
function hello(result) {  
  alert(result)  
}
```

Here, a variable with the name `result` is declared. The parameter, which is passed to the function when it's called, determines in this case whether the variable in question is a text or numeric variable.

Local Variables

A variable like `result` in the foregoing example is local. This means that it only exists in the function `hello`, not in other functions that may be declared later. If you'd just like to reserve a space in memory in which a value or some text can be stored, something you'd like to read out later, then you should declare a variable that's not a parameter. Here's another option:

```
function hello() {  
  var result  
  result="Here I am!"  
  alert(result)  
}
```

This function works entirely without parameters. During testing, you'll need to take the parameter out of the function call. The function declares the variable `result`, assigns it the text `Here I am!` and displays that text. One characteristic of JavaScript is that variables can be assigned values right when they're declared. In the following example you'll see how numbers can be the contents of variables too.

```
function hello() {  
  var eggs=5  
  alert("Today I bought " + eggs + " eggs.")  
}
```

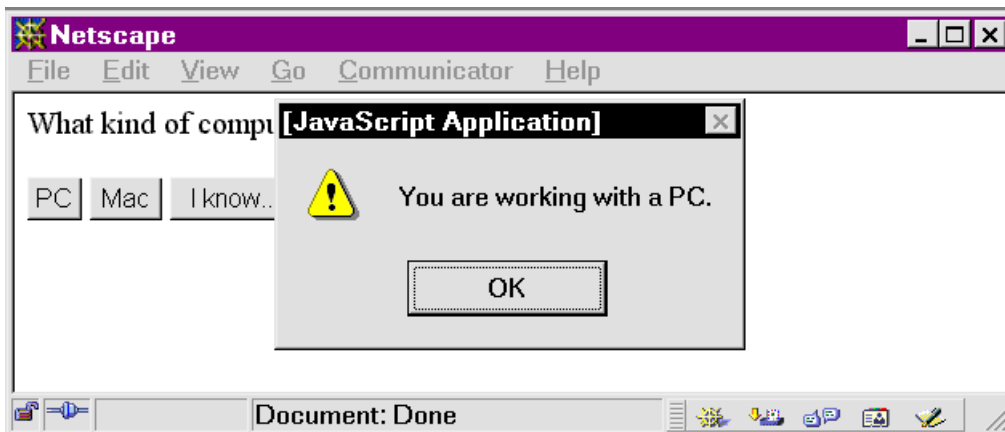
Here, a variable `eggs` is declared, which is assigned the value `5`. After that, the text `Today I bought 5 eggs.` is displayed, which makes use of the variable.

Global Variables

All uses of variables that you've seen so far have the disadvantage that they can only be used in the one function in which they were declared. Let's pose the following practical problem for ourselves: an HTML page is supposed to contain three buttons. Users should click the first button if they are using a PC, and the second one if they are using a Mac. A click on the third button is supposed to produce a message that tells the user which kind of computer they are using. From the programmer's point of view, this means that if the user clicks the first button, the text "PC" should be stored. If the user clicks the second button, "Mac" should be stored. If the user clicks the third button, the browser should display an appropriate message. The source code looks like this:

```
<html>
<head>
<script language="JavaScript">
<!--
var computer="unknown"
function message(){
alert('You are working with a ' + computer + '.')
}
//-->
</script>
</head>
<body>
What kind of computer are you using?
<form><input type="button" value="PC" onClick="computer='PC'">
  <input type="button" value="Mac" onClick="computer='Mac'">
  <input type="button" value="I know..." onClick="message()"> </form>
</body>
</html>
```

In the header, the global variable `computer` is declared, which is assigned the value `unknown`. The user clicks a button. The kind of computer will be written to the variable. When the user clicks the third button, an appropriate message will be displayed. The result looks like this:



Mathematical Operations

In addition to variables and functions, mathematical operations are another basic element of JavaScript programming. But this isn't complicated at all. Here I'll just explain the basic kinds of calculation and a few peculiarities. If you want to carry out a calculation, you'll need a variable to which you can write the result – here we'll call this variable `result` – and in most cases you'll also need two variables that should be added together (here: `a` and `b`). All operations can use whole numbers and decimal remainders. There's just one point to remember about decimals:

Addition: `result = a + b`

Subtraction: `result = a - b`

Multiplication: `result = a * b`

Division: `result = a / b`

Whole number rounding: `a = Math.round(b)`

There are also operations that require the use of just one variable: addition of 1 to a variable (this is functionally equivalent to `result = result + 1`):

`result++`

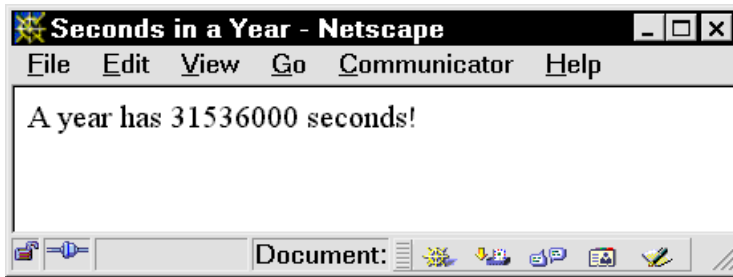
Subtraction of 1 from a variable (this is functionally equivalent to `result = result - 1`):

`result--`

Anyone can see how these operations work. But nevertheless, I'm going to provide a practical example so that you can test other kinds of calculations than the multiplication shown in the source code. This program is supposed to display the number of seconds in a year. Naturally we don't just want to plop the number into the HTML file; rather, we'd like to give the number of days in the year, times the number of hours in a day, times the number of minutes in an hour, times the number of seconds in a minute. We'll display this result. Here's the source code:

```
<html>
<head>
<title>Seconds in a Year</title>
</head>
<body>
A year has
<script>
var seconds=365*24*60*60
document.write(seconds)
</script>
seconds!
</body>
</html>
```

The result will look like this in your browser:



Given such a simple example, one could, of course, just output the sentence “A year has 3156000 seconds!” in the HTML file. Later on, we’ll see examples that are calculated using values input by the user of the page.

Repeated Performance

In every programming language, there are times where you’ll want to repeat the same command over and over. The commands for repeated performance are the same in nearly every language: they are called **for** and **while**. Here you’ll see examples of each command.

Looping with for

Let’s assume that you want to publish a table of squared numbers from 1 to 100. Now you could, of course, calculate all the values with your pocket calculator and then enter them into a table, or you could call on JavaScript to help you.

For the JavaScript variation – which requires significantly less source code – you’ll need the following function:

```
function square() {  
  for (var i=1; i<=100; i++){  
    document.write("<tr><td>")  
    document.write(i)  
    document.write("</td><td>")  
    document.write(i*i)  
    document.write("</td></tr>")  
  }  
}
```

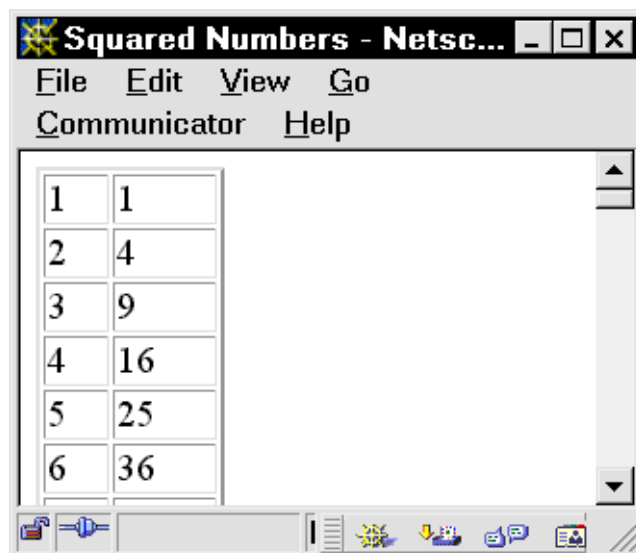
The command `for (var i=1; i<=100; i++){` opens the loop. `for` (is mandatory. Since we haven’t defined any counter variables yet, `var i` comes next. If the variable had already been declared, then you’d just need `i` here. With `=1`, this variable is given its starting value, then comes a semicolon. In most cases, the counter variable should, as in the foregoing example, be incremented by one each time the loop runs – this is what `i++` is for. The curly bracket marks the beginning of the loop. Then come the commands that should be repeated. Here the first command opens a table line and a table cell. Then the counter variable is written to the table, the cell is closed, and a new one is opened. The square of the counter variable is written to the table, and finally the table cell and line are closed. The curly bracket marks the end of the loop.

This function does the following: for all numbers from 1 to 100, it creates a line. In the first

column, it writes the number; in the second column, it writes the number's square. Naturally then you have to open a table in the HTML file. The whole source code looks like this:

```
<html>
<head>
<title>Squared Numbers</title>
<script language="JavaScript">
<!--
function square(){
for (var i=1; i<=100; i++){
document.write("<tr><td>"
document.write(i)
document.write("</td><td>"
document.write(i*i)
document.write("</td></tr>"
}
}
//-->
</script>
</head>
<body>
<table border=2>
<script>
square()
</script>
</table>
</body>
</html>
```

In the browser, this example should look as follows – remember that the loading of the table can take some time on older computers due to its size:



Looping with while

The `while` loop closely resembles the `for` loop. But the `while` loop doesn't require any counter variables and there's no operation that changes the value of these counter variables. The `for` loop will simply run until the given condition is false. Its beginning is nearly identical to that of the `for` loop – it is used when you already have a counter variable with a value since no fixed value is given to the counter variable at the beginning of the loop. As an example, let's use the squaring program again. This time, the function will look like this:

```
function square() {
  i=45
  while (i<=100){
    document.write("<tr><td>")
    document.write(i)
    document.write("</td><td>")
    document.write(i*i)
    document.write("</td></tr>")
    i++
  }
}
```

At the beginning of the function, the value of the counter variable is set to 45. Later on, this initial value can be read off of a form, for example. In the parentheses of the `while` command comes the condition that must be fulfilled for the loop to end. Since otherwise the value of the counter variables would remain constant, it must be changed in the `while` loop. This is what the `i++` command does. The whole source code looks as follows – the result in the browser resembles that in the last example, but the list begins with 45:


```
<html>
<head>
<title>Squared Numbers</title>
<script language="JavaScript">
<!--
function square(){
i=45
while (i<=100){
document.write("<tr><td>")
document.write(i)
document.write("</td><td>")
document.write(i*i)
document.write("</td></tr>")
i++
}
}
//-->
</script>
</head>
<body>
<table border=2>
<script>
square()
</script>
</table>
</body>
</html>
```

When using `for` and `while` loops, it's important that the programmer look out for so-called endless loops. These can occur if the value of the counter variable never changes, if it's set equal to the beginning value by a mathematical operation, or if it's set to a fixed value within the loop. Such endless loops can only be exited by "strong-arming" your computer; try Ctrl + Alt + Del on the PC or Command + Alt + Esc on the Mac if your browser stops reacting. Sometimes, you'll even need to reboot your computer in order to restart your browser. This shows that you need to be really careful when programming loops, for if the browser crashes when you're looking at your site online, then you'll be a sad surfer for a while.

Conditional Operations

A further important element of JavaScript is conditional operation, the `if` command. In the course of this booklet you'll see that `if` is one of the most-used commands. The following example should illustrate this.

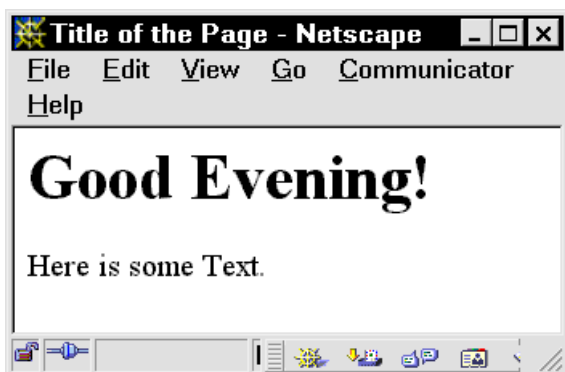
Depending on the time of day, a surfer on your Web site should be greeted differently – at 7:00 PM, for example, the message should be “good evening.” The following source code is necessary to accomplish this:

```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function greeting() {
var time=new Date()
var hours=time.getHours()
var hello="Hello, Nightowl!"
if (hours>5){
hello="Good Morning!"
}
if (hours>12){
hello="Good Day!"
}
if (hours>16){
hello="Good Evening!"
}
if (hours>22){
hello="Good Night!"
}
document.write(hello)
}
//-->
</script>
</head>
<body>
<h1>
<script>
greeting()
</script>
</h1>Here is some Text.
</body>
</html>
```

The first two lines of the function store the number representing the current time in the variable `hours`. This functions exactly as in the “What time is it?” example at the beginning of this booklet. Next, a variable `hello` with the greeting “Hello, Nightowl!” is declared. Next come the `if` statements. The introductory `if` is mandatory. After that, in parentheses, come the conditions upon fulfillment of which the commands in the `if` section will be carried out. In our example, this greeting should be issued if it’s 6:00 PM or later. The curly bracket marks the beginning of the commands in the `if` section. In the next line, the greeting is set to “Good Morning!” and the curly bracket ends the `if` command.

What happens now? First, the greeting is set to “Hello, Nightowl!” If it’s 6:00 AM or later, the greeting is set to “Good Morning!” If it’s already after 1:00 PM then it’s set to “Good Day!,” etc.

Finally the greeting is displayed. Since I’ve worked through the whole night yet again, I’ll be greeted as follows by the browser:



This program can be taken over as-is into a Web site. It’s not especially resource-intensive, that is, it won’t disturb animations, but it makes an individualized impression.

Standard Objects

An object is a collection of functions and variables. Standard objects are predefined in JavaScript. The already familiar `document.write("Bla bla")` command is, for example, a function of the object `document`. The command `alert("Bla bla")` is a function of the `window` object, since it's really an abbreviation for `window.alert("Bla bla")`. Yet unfamiliar is `document.bgColor`, which is a variable of the `document` object. The function of a variable of a standard object is always designated by the name of the object, a dot, and the name of the function or variable.

There are a multitude of standard objects; they are far too numerous to discuss here. Therefore, we'll discuss just the most important ones.

document

In the `document` object, you'll find all functions and variables that have to do with HTML documents. But since this object too contains more functions and variables than we can discuss here, we'll limit ourselves in the following to the most important ones:

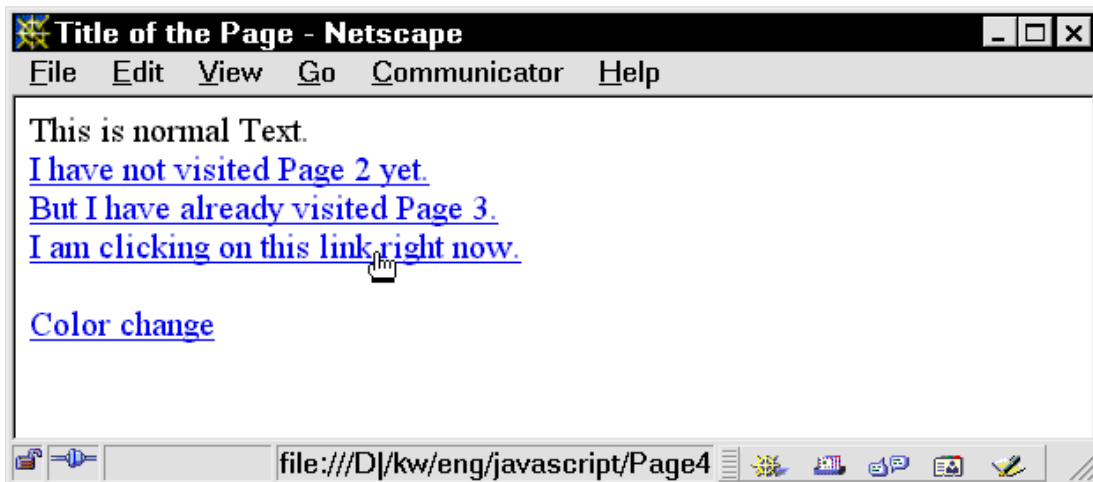
Colors in the Document

Using JavaScript, you can access the color settings of HTML documents. Elements you can change include the text color, background color, and the colors of not yet visited, already visited, and active links. Unfortunately, access to these colors is not always reliable in all browser versions. Netscape in particular leaves something to be desired. In the following practical example, the various colors are supposed to be changed with a click of the mouse. If you're using Netscape, only the background color will change – but Microsoft Internet Explorer isn't always reliable either. In practice, you should only use the color-manipulation capabilities of JavaScript to change the background color. What follows is the source code for the abovementioned example:

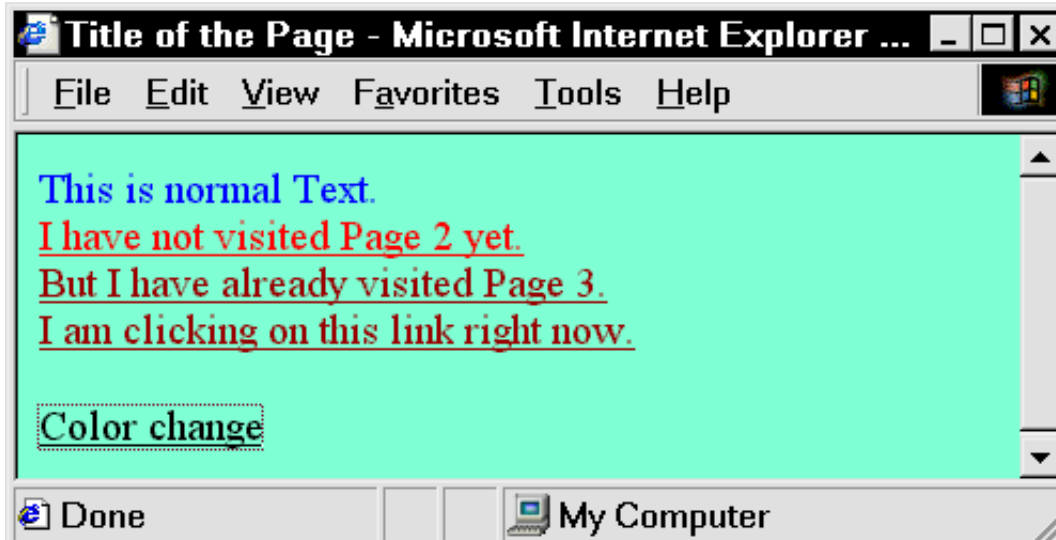
```
<html>
<head>
<title>Title of the Page</title>
<script language="JavaScript">
<!--
function color(background, foreground, new_link, visited_link,
active_link) {
document.bgColor=background
document.fgColor=foreground
document.linkColor=new_link
document.vlinkColor=visited_link
document.alinkColor=active_link
}
//-->
</script>
</head>
<body>
This is normal Text.<br>
<a href="Page2.htm">I have not visited Page 2 yet.</a><br>
<a href="Page3.htm">But I have already visited Page 3.</a><br>
<a href="Page4.htm">I am clicking on this link right now.</a><p>
<a
```

```
href="javascript:color('7FFFD4','0000FF','FF0000','990000','000000')
">Color change</a>
</body>
</html>
```

Before you click on the “Color change” link, the whole thing should look like this if you’re using Netscape:



Because Microsoft Internet Explorer functions a bit better, what I’m showing you here is the result after clicking on “Color change” in this browser:

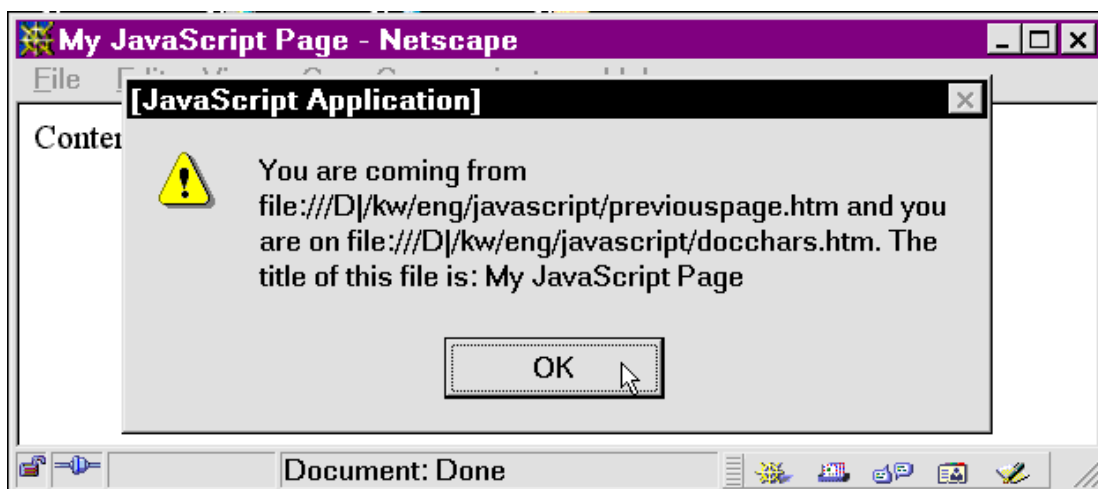


Document Properties

There are three text constants in an HTML file that can only be read out by JavaScript but not changed. These are the URL of the file, the address from which the current file is read (`document.referrer`), the URL address of the file itself (`document.location`), and the title of the current document (`document.title`). The following example issues a message upon loading, which contains these three constants:

```
<html>
<head>
<title>My JavaScript Page</title>
<script language="JavaScript">
<!--
function message() {
alert("You are coming from " + document.referrer + " and you are on
" + document.location + ". The title of this file is: " +
document.title)
}
//-->
</script>
</head>
<body onLoad="message()">
Contents of the Page
</body>
</html>
```

In the browser, the message will appear as follows:



This application doesn't appear very useful at first glance – but scripts of this kind can be useful if, for example, on your Web site you'd like to determine whether the user is coming from another site or your own.

Pictures in a Document

If you surf regularly on the Internet, you've probably noticed a much-loved feature: links in the form of pictures that change their color or something when you pass over them with the mouse. The secret to the color change is that the picture behind which the link lies is replaced by another picture. The standard object `document` is responsible for administering these pictures. For each pictures that you'd like to have produce this effect, you have to create a name in the HTML source code. This could go like this:

```

```

Here you've created the variable `document.picture.src`, which is assigned the graphic `picture.gif`. Now you need to create a second picture of the same size, which should replace the first picture upon mouse contact. The value of `document.picture.src` can be changed, whereupon the picture will be switched. You could proceed quite simply and assign it another picture file, for example, as follows:

```
document.picture.src="picture2.gif"
```

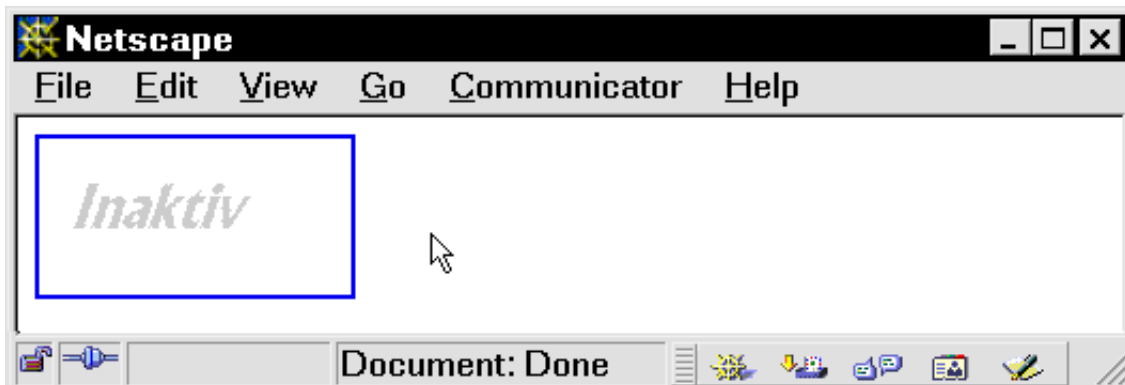
This might work on your PC at home – but common practice on the World Wide Web suggests that this is problematic: the browser will only produce the second picture if the first picture is touched with the mouse. But clever programmers have found a solution to this problem: upon loading the document, a new picture is generated that isn't displayed in the browser. You can do this with the following code:

```
substitution=new Image  
substitution.src="picture2.gif"
```

To retain the original picture in memory, treat it exactly the same way. The entire source code looks like this:

```
<html>  
<head>  
<script language="JavaScript">  
<!--  
picture1=new Image  
picture1.src="picture1.gif"  
picture2=new Image  
picture2.src="picture2.gif"  
//-->  
</script>  
</head>  
<body>  
<a href="Page2.htm" onMouseOver="document.picture.src=picture2.src"  
onMouseOut="document.picture.src=picture1.src">  
</a>  
</body>  
</html>
```

Several explanations of the source code are necessary here. Upon loading the page, the global JavaScript source code is carried out. It loads the picture data from `picture1.gif` and `picture2.gif` as described above. Upon contact with the mouse (`onMouseOver`), the second picture is displayed, upon leaving (`onMouseOut`), the first one appears again. The whole thing looks like this in the browser if the mouse is not touching the link:



If you pass the mouse over the picture, you'll see the following picture:



This application can also be taken over as-is into Web sites. If you want to place several changing pictures on a page, you must, of course, choose different names for each picture. On the other hand, as in the following example, a picture can appear in more than one place. An arrow is supposed to show over which link on a list the mouse is at the moment. To accomplish this, you'll need a table with two columns and, in our case, three lines. In the right-hand cells, you'll find pictures that have various names and to which, at first anyway, the same picture file is assigned. In the left-hand cells, you'll create links, which, when there is contact with the mouse, will change the picture as in the foregoing example. This time, however, a picture should be changed behind which there isn't any link. For this purpose you'll just need to replace the picture name of another picture – in our example, the name of the picture that's in the neighboring cell. So that you'll understand this procedure, I'll show you the two pictures `active.gif` and `inactive.gif`, which must be the same size.



`active.gif`

`inactive.gif`

Here's the source code:

```
<html>
<head>
<script language="JavaScript">
<!--
picture1=new Image
picture1.src="inactive.gif"
picture2=new Image
picture2.src="active.gif"
//-->
</script>
</head>
<body>
<table border=0 cellpadding=0 cellspacing=0><tr><td><a
href="page2.htm" onMouseOver="document.Link1.src=picture2.src"
onMouseOut="document.Link1.src=picture1.src">To Page 2</a></td>
<td></td></tr>
<tr><td><a href="page3.htm"
onMouseOver="document.Link2.src=picture2.src"
onMouseOut="document.Link2.src=picture1.src">To Page 3</a></td>
<td></td></tr>
<tr><td><a href="page4.htm"
onMouseOver="document.Link3.src=picture2.src"
onMouseOut="document.Link3.src=picture1.src">To Page 4</a></td>
<td></td></tr></table>
</body>
</html>
```

The principle is the same as in the previous example, where I explained how this works. Here you can see what Netscape makes out of this:



document.frames

`document.frames` is really an object of the object `document`, but since it can be used independent of it, it will be covered in a separate section. The object `document.frames` is used almost exclusively to change several frames with one click of the mouse. Let's take a look at a practical example:

Our goal here is to create a frameset that will divide the screen vertically into two pieces. The right piece will then be divided again. The left frame should display a table of contents in which various animal names can be clicked. If, for example, the user clicks on "elephant," the lower right corner of the frame should display information about elephants. If, by contrast, the user clicks on "crocodile," then the frame should display "reptiles" and "crocodile."

To carry out this project, first we'll need a frameset that we'll call `animalinfo.htm`:

```
<html>
<head>
<title>Animal Information</title>
</head>
<frameset cols="150,*">
<frame src="contents.htm">
<frameset rows="100,*">
<frame src="kindofanimal.htm">
<frame src="startanimal.htm">
</frameset>
</frameset>
</html>
```

`kindofanimal.htm` – which needs no title since it's displayed on the frame – looks like this:

```
<html>
<head>
</head>
<body>
<h1>Kind of Animal</h1>
</body>
</html>
```

Then there's the file `startanimal.htm`:

```
<html>
<head>
</head>
<body>
Here is where some information about animals will appear.
</body>
</html>
```

Then you'll need to create three files, which are only a little bit different than `kindofanimal.htm`: `mammals.htm`, `reptiles.htm`, and `fish.htm`. Here you should replace the text `Kind of Animal` with `mammals`, `reptiles`, or `fish`, respectively. In addition you'll need some "animal files." To create these, just modify the file

`startanimal.htm`. Replace the text with information about an animal and name the file after the animal in question. I've created `mouse.htm`, `elephant.htm`, `crocodile.htm`, `lizard.htm`, `greatwhiteshark.htm` und `goldfish.htm`

And now to the file `contents.htm`, which contains the JavaScript code:

In this file, you'll find links that reference JavaScript functions. We've already seen this – the references look like this:

```
<a href="javascript:change_frames('mammals.htm','elephant.htm')">
Elephant</a>
```

In the JavaScript portion of the file, the following functions must be declared:

```
function change_frames(file1,file2){
parent.frames[1].location.href=file1
parent.frames[2].location.href=file2
}
```

The explanation: the frameset is constructed like a tree. The main file – in our case `animalinfo.htm` – is the trunk of the tree, the frames are the branches. In order to reach one branch from another, you have to climb up the tree trunk. In JavaScript: to change from one frame to another, you have to use the main file. The reference link `parent` is responsible for this. To “branch off” to another frame, you have to give its index. When you declare a frameset, the browser begins counting at 0, so that the left-hand frame in our example is assigned the index 0, the frame for the kind of animal the index 1, and the lower right-hand frame the index 2. In our case, then, frame 1 and frame 2 must be changed. You can access the filenames using `location.href`. In the function, therefore, the two right-hand frames are assigned different file names at the same time; the file names are passed to the parameters `file1` and `file2`.

The entire source code for the file `contents.htm` looks like this:

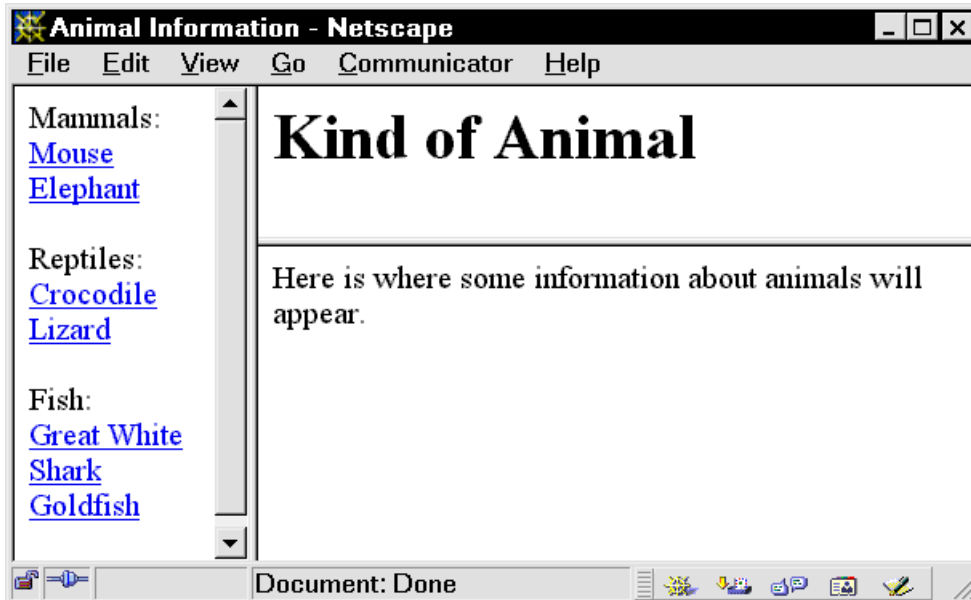
```
<html>
<head>
<script language="JavaScript">
<!--
function change_frames(file1,file2){
parent.frames[1].location.href=file1
parent.frames[2].location.href=file2
}
//-->
</script>
</head>
<body>
S&auml;mammals:<br>
<a href="javascript:change_frames('mammals.htm','mouse.htm')">
Mouse</a><br>
<a href="javascript:change_frames('mammals.htm','elephant.htm')">
Elephant</a><p>
Reptiles:<br>
<a href="javascript:change_frames('reptile.htm','crocodile.htm')">
Crocodile</a><br>
<a href="javascript:change_frames('reptile.htm','lizard.htm')">
Lizard</a><p>
Fish:<br>
<a href=
"javascript:change_frames('fish.htm','greatwhiteshark.htm')"> Great
```

```

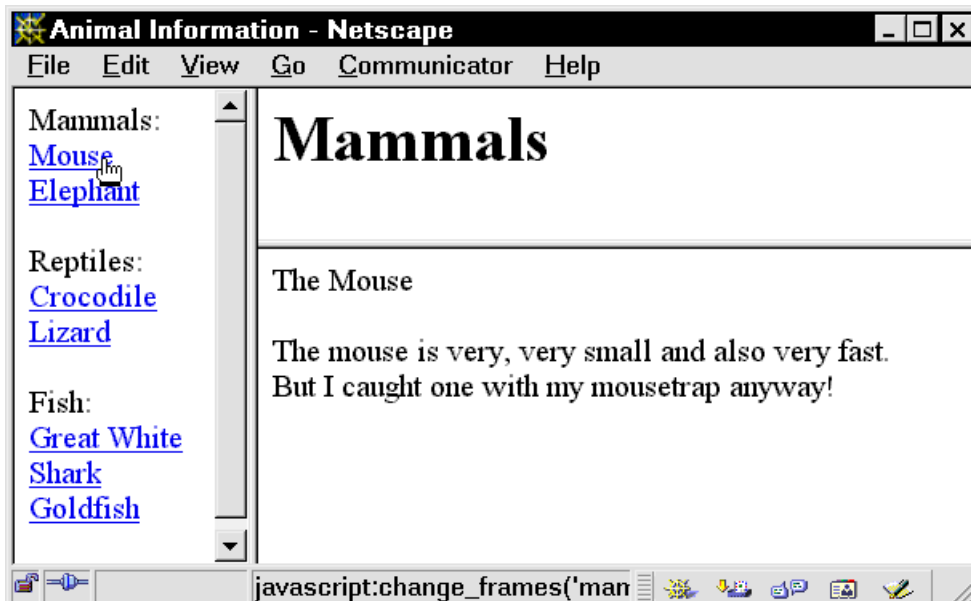
White Shark</a><br>
<a href= "javascript:change_frames('fish.htm','goldfish.htm') " >
Goldfish</a><p>
</body>
</html>

```

Upon loading the file `animalinfo.htm` into the browser, you should see this:



If you click the mouse on "Mouse," you should see the following picture:



document.forms

Like frames, each form that's a part of an HTML file gets an index. Again, the browser starts counting with 0; as a rule, only one form with the index 0 is present. The object `forms` has several properties: `document.forms[0].length`, for example, returns the number of entry fields on the form with index 0. But this object is hardly used. More important is a "secondary object" of `document.forms`, namely `document.forms[0].elements`, where `elements` is replaced with the name of an element. With the help of this object, it's possible to read out the entry fields on a form and overwrite them. This requires some theoretical knowledge, specifically with respect to how the various element types are addressed with different references. All elements in HTML source code are given names – in the following example, `Elementname` stands for the actual element names.

Text Entry Fields

The following property can read out or overwrite the current text in an entry field:

```
document.forms[0].Elementname.value
```

This next property can read out or place default text in an entry field:

```
document.forms[0].Elementname.defaultValue(DefaultValue)
```

The following function marks the text in an entry field:

```
document.forms[0].Elementname.select()
```

And the following function, which places the cursor in a field, is defined for all entry fields, but used almost exclusively for text entry fields:

```
document.forms[0].Elementname.focus()
```

Radio and Check buttons

This property reads out or writes whether a radio button has been selected. The index of the radio button is indicated here with `n` – as always, the browser always begins counting with 0. Possible values for this property are `0` (not selected), `1` (selected), or, if you prefer, `false` (not selected) and `true` (selected).

```
document.forms[0].Elementname[n].checked
```

With this property you can read out and write the default setting of a radio button (index and possible values as above):

```
document.forms[0].Elementname.defaultChecked
```

Check buttons have the same properties; here `n` is the index of an element if there are several elements with the same `name` parameter present in a group.

Drop-Down Lists

Drop-down lists have a property with help of which you can read out or write whether an option is selected (index and possible values as above):

```
document.forms[0].Elementname.options[n].selectedIndex
```

The `value` value can be read and written:

```
document.forms[0].Elementname.options[n].value
```

Pizza Service

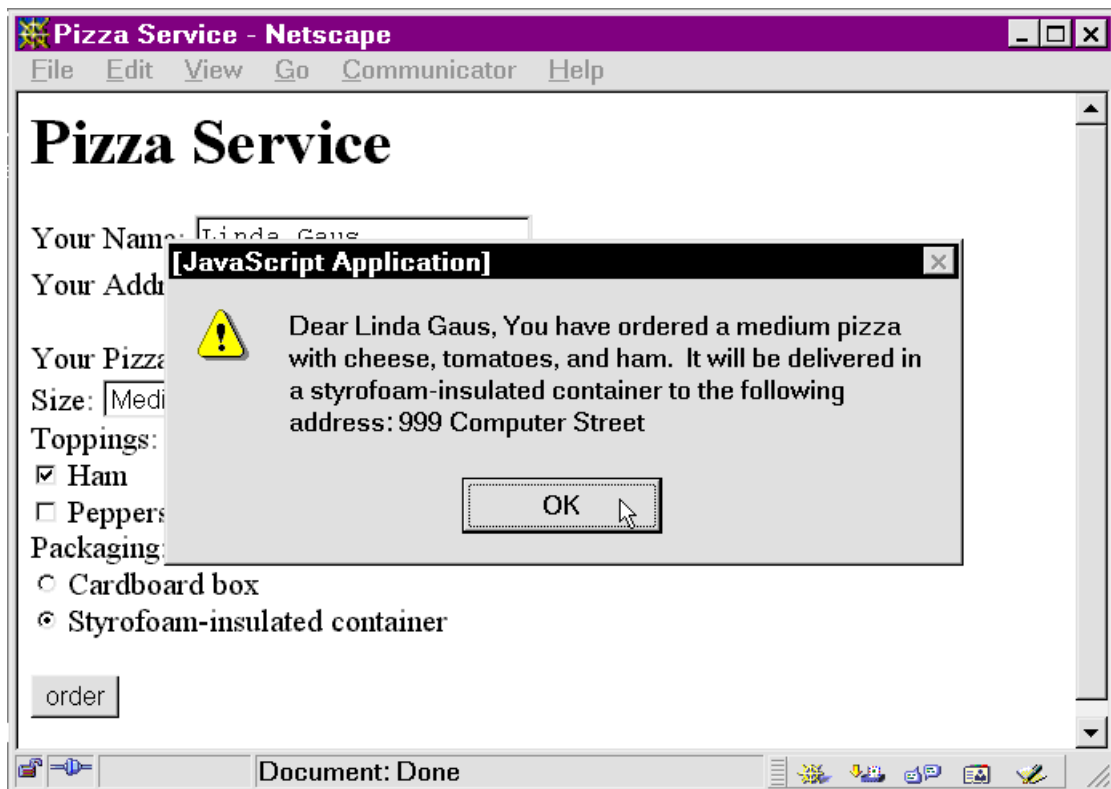
The following example is intended to explain the function of the `document.forms` object. Here, we're concerned with ordering a pizza that can have various characteristics (size, toppings). Upon submitting the form, a message is generated that tells the hungry person what kind of pizza he or she has just ordered. Here is the long and complete source code for this application:

```
<html>
<head>
<title>Pizza Service</title>
<script language="JavaScript">
<!--
function confirmation() {
var result="Dear "
result = result + document.forms[0].Name.value
result = result + ", You have ordered a "
if (document.forms[0].size.options[0].selected) {
result = result + "small"
}
if (document.forms[0].size.options[1].selected) {
result = result + "medium"
}
if (document.forms[0].size.options[2].selected) {
result = result + "large"
}
result = result + " pizza with cheese, tomatoes, "
if (document.forms[0].topping[0].checked) {
result = result + "and ham. "
}
if (document.forms[0].topping[1].checked) {
result = result + "and peppers. "
}
result = result + " It will be delivered in a "
if (document.forms[0].packaging[0].checked) {
result = result + "cardboard box"
}
if (document.forms[0].packaging[1].checked) {
result = result + "styrofoam-insulated container"
}
result = result + " to the following address: "
result = result + document.forms[0].address.value
alert(result)
}
//-->
</script>
</head>
<body>
<h1>Pizza Service</h1>
<form action="mailto:gaushaus@bellatlantic.net" method="post">
```

```
onSubmit="confirmation() ">
Your Name:
<input name="Name" size=20><br>
Your Address:
<input name="address" size=20><p>
Your Pizza:<br>
Size:
<select name="size">
<option>Small
<option selected>Medium
<option>Large
</select><br>
Toppings:<br>
<input type=checkbox name="topping">Ham<br>
<input type=checkbox name="topping">Peppers<br>
Packaging:<br>
<input type=radio name="packaging">Cardboard box<br>
<input type=radio name="packaging">Styrofoam-insulated container<p>
<input type=submit value="order">
</form>
</body>
</html>
```

The HTML portion of the document consists of a normal entry form, whose fields contain names. In the `<form>` tag there's an e-mail address to which the completed form will be sent – I'd ask you kindly to change the address, lest my translator find the pizza wishes of all my readers in her e-mailbox. The form uses the event handler `onSubmit` in conjunction with the function `confirmation`, which is called when the form is submitted.

In the function, a variable with the contents `Dear` is declared. This variable is eventually expanded to include a result sentence. As described above, the name is added, then the text “ , **You have ordered a**”. Depending on what the user inputs, the size, toppings, and packaging of the pizza are filled in as indicated. The whole variable, which now contains a result sentence, is displayed as a message using the `alert` command. This pizza order looks as follows if you're using Netscape:



You can expand the pizza entry form to include more toppings and packaging options if you'd like to understand the `document.form` object better. You can also add a further entry field for the recipient's address, which can be output in the message. You can even add two radio buttons for normal/express delivery.

Euro Calculator

In connection with the `form` object, here's another good example of how you can use JavaScript to write to text entry fields. Let's build a Euro calculator, which will convert all currencies of the Euro countries into Euros and vice versa. Here, without further ado, is the source code. The JavaScript function `calculate` is left out, but must be added later.

```
<html>
<head>
<title>Euro Calculator </title>
<script language="JavaScript">
<!--
var a=0
function calculate(){ ...
}
//-->
</script>
</head>
<body>
<h2>Euro Calculator </h2>
<form>
<input name=amount size=10 onChange="a=0">
<select name="currency" onChange="calculate()"
onFocus="calculate()">
<option value="13.7603">ATS</option>
<option value="40.3399">BEF/LUF</option>
<option value="1.95583">DEM</option>
<option value="166.386">ESP</option>
<option value="5.94573">FIM</option>
<option value="6.55957">FRF</option>
<option value="0.787564">IEP</option>
<option value="1936.27">ITL</option>
<option value="2.20371">NLG</option>
<option value="200.482">PTE</option>
</select>
corresponds to
<input name=euro size=10 onChange="a=1">
Euros.<br>
<input type=button value="Calculate" onClick="calculate()">
</form>
</body>
</html>
```

In the HTML file, a form is created that has two text fields, one for the amount in Euros and one for the other currency, a field for calculating, and a selection field – we'll get to the `onChange` commands in a moment. The selection field contains all currencies that can be converted into Euros. The fixed conversion rates are the values for `value`.

When you click the Calculate button or change currencies, reference is made to a JavaScript function that we'll call `calculate`. Its purpose is to convert the amount that was changed last – be it Euro or other currency. Now the meaning of the `a` variable should become clear: each

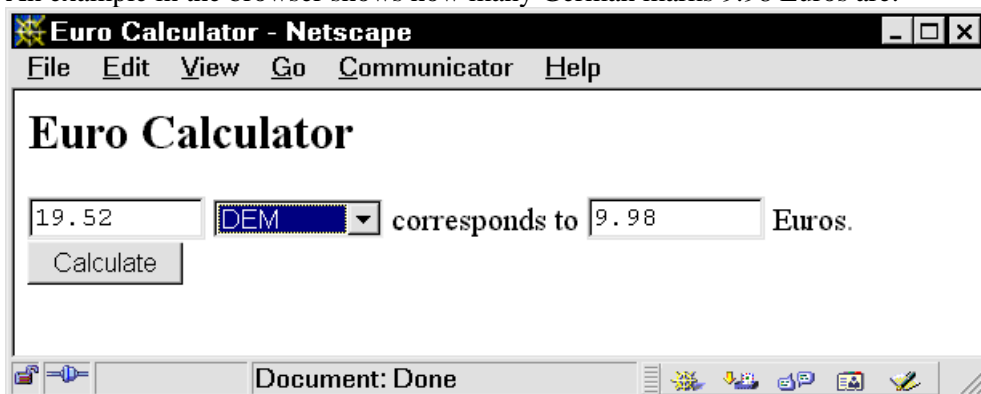
time that the text entry field is changed, the value of this variable changes too. If the Euro value is changed, it is set to 1; otherwise, it is set to 0. The function needs this variable to know which value was changed last, and thus, which value should be converted. The function looks like this:

```
function calculate() {
  if (a==0) {
    document.forms[0].euro.value=Math.round(document.forms[0].amount.value / document.forms[0].currency[document.forms[0].currency.selectedIndex].value * 100) / 100
  }
  if (a==1) {
    document.forms[0].amount.value=Math.round(document.forms[0].euro.value * document.forms[0].currency[document.forms[0].currency.selectedIndex].value * 100) / 100
  }
}
```

How it works: if **a** has a value of 0, then the Euro entry field is used. **Math.round** rounds a number to the nearest whole number. The value in the first text entry field is rounded, divided by the **value** value of the text entry field option that's currently selected (which corresponds to the conversion rate), and multiplied by 100. Finally, the whole thing is divided by 100. The multiplication by 100, which is rounded and then divided again by 100, is necessary to produce a result with two decimal places.

If **a=1**, then the Euro entry is multiplied by the conversion rate – this part of the function works analogously.

An example in the browser shows how many German marks 9.98 Euros are:



Strings

In computer language, a string is a series of characters. In JavaScript there are a few useful commands with help of which you can change stored strings or read the strings out in changed form.

The String Object

Here I'll introduce you to the most important of the commands that belong to the string object:

length

This property is used to read out the number of characters in a string. `xy.length` has as its value the number of characters in the variable `xy`. For example, if this string contains the text "Count my numbers!", then the value is 17.

substring

This function returns a portion of a string. If you would like to write the first 10 characters in the string `xy` into a variable `z`, for example, then the command would be `z=xy.substring(0,9)`, because the computer begins counting with 0.

toLowerCase

This method changes all the letters in a string into lower case. If, for example, the text "I am HERE!" is stored in the variable `xy`, you would need to issue the following command to transform the string into "i am here!": `xy=xy.toLowerCase`

toUpperCase

This function operates analogously, except that it changes lower-case letters into upper-case ones. If you'd like to store the contents of the variable `xy` in the variable `z` in upper-case letters, then you'd need to issue the following command: `z=xy.toUpperCase`

Moving Text

Surely you've been asking yourself what all of these string functions are good for. An example would be the following program, which displays moving text in the status line. The code looks like this:

```
<html>
<head>
<title>Moving Text</title>
<script language="JavaScript">
<!--
var text="The Yankees beat the Mariners 8-2!"
var result="start"
function movingtext(){
if (result=="start"){
for (var i=1; i<=140; i++){
text=" "+text
}
result=text
}
result=result.substring(1,result.length)
```

```

window.status=result
if (result==""){
result=text
}
setTimeout("movingtext()",150)
}
//-->
</script>
</head>
<body onLoad="movingtext()"; return true>
<h2>Have a look at the status line!</h2>
</body>
</html>

```

And here's how the program works: a global variable `text` is declared, to which the moving text, which will appear later, is assigned. In addition, another variable is declared, which we've called `result` and whose value is initially set to `start`. When the document is loaded, the function `movingtext` is called up by the `onLoad` reference in the body tag. Here it is established whether `result` contains the string `start`.

This is the case the first time the function is called, so the following loop will run 140 times. 140 empty spaces are set in front of our original text. The text is always displayed left-aligned in the status line. To make it move to the right, the text must be preceded by leading empty spaces, which are gradually taken away. The most convenient number of empty spaces depends on the resolution of the user's screen – you'll probably need to find an appropriate compromise. The variable `result` is assigned the string stored in the variable `text`. From here on, the variable `text` is not changed, since, thanks to the changed value of `result`, the `if` statement is omitted in later function calls. The first character of the variable `result` is deleted – or actually it is overwritten with the second through last characters in the string. Now the result is written to the status line.

If all the characters have already been deleted, then `result` is devoid of content and is set back to its starting value, which is still stored in `text`, by means of an `if` statement. The command `setTimeout("movingtext()",150)` is still unfamiliar to you – it informs the browser that the function `movingtext` should be called again after 150 milliseconds.

After this, the function ends. 150 milliseconds later, it is started by the browser again, the result is truncated by one character, and the whole process is repeated until the result is empty, whereupon `result` is set back to its original value. And so the little game repeats itself over and over until another HTML page is loaded.

In the browser, the whole thing looks like this:



This effect should not be built into sites that strive to make a serious impression – in that kind of context, this effect is shunned. On private or entertainment-related pages, however, it works quite well for news or the “joke of the day.”

User-Defined Objects

The goal of this section is best attained through presentation of a concrete example. The names and ages of several colleagues in a particular firm should be stored in variables. To do this, you can either declare lots of variables (`smith-name`, `smith-age`, `jones-name`, `jones-age`, etc.) or you can create objects. Doing the latter will require the following function:

```
function colleague(name,age) {
  this.name=name
  this.age=age
}
```

With the command `var smith=new colleague("Michael Smith",43)`, the function is called – a new object is created and the name and age are assigned. After this, you can reference the name with `smith.name` and the age with `smith.age`.

In addition, you can declare your own objects with the following command:

```
smith=new Object
```

Then you can freely assign properties, for example, like this:

```
smith.firstname="Michael"
smith.lastname="Smith"
smith.hobby="soccer"
```

Arrays

An array is a series of variables of the same type, which are referenced with the same name and an index.

The following command declares a new array – `new` can be replaced with another name (see reserved words on page 58).

```
new = new Array
```

Here, you are declaring a whole series of variables, namely `new[1]`, `new[2]`, `new[3]`, etc. They must all be of the same type, that is, `new[1]` can't be a number if `new[2]` is a string.

You'll find an example of the application of user-defined objects and arrays later on in the "Quiz" section.

Working with Frames

All of the foregoing examples, save for the animal information program, restrict themselves almost entirely to one HTML page. In more complex applications, however, variables often need to be made available across several HTML pages. Normally, server-side scripts are required for this, but you'll find that server-side scripts aren't allowed by many large providers like AOL. If you're clever, though, you'll be able to accomplish many of the same tasks with JavaScript. This requires working with frames. The main file, which isn't even visible since it contains only references to subfiles, contains the relevant variables. The individual frames can be changed, but the variables in the main file remain. This may sound rather abstract, so here's a brief example:

The main file called `FRAMES.HTM` divides the browser window horizontally into two portions; in the JavaScript portion, a global variable `name` is declared:

```
<html>
<head>
```

```
<title>Frames</title>
<script language="JavaScript">
<!--
var name
//-->
</script>
</head>
<frameset rows="100,*">
<frame src="advertising.htm">
<frame src="start.htm">
</frameset>
</html>
```

The file **ADVERTISING.HTM**, which contains an advertising banner, looks like this:

```
<html>
<head>
</head>
<body>
<center>

</center>
</body>
</html>
```

The source code of the other file in the frameset, namely the file **START.HTM**, looks like this:

```
<html>
<head>
<script language="JavaScript">
<!--
function GoOn() {
parent.name=document.forms[0].namefield.value
document.location.href="page2.htm"
}
//-->
</script>
</head>
<body>
<h2>Welcome to my Homepage!</h2>
Please enter your name:<br>
<form>
<input name="namefield">
<input type="button" value="Continue" onClick="GoOn()">
</form>
</body>
</html>
```

The file contains a form for entering a name and a “Continue” button. If the button is clicked, the function `continue` is called. It supplies the variable in the next frame, that is, in **FRAMES.HTM**, with the contents of the entry field. Here’s a brief overview:

`parent.` branches off to the next frame up.

`top.` branches off to the highest frame in the browser.

`frames[n].` branches off to the next level down in the frame with index `n` (see also the animal

information program).

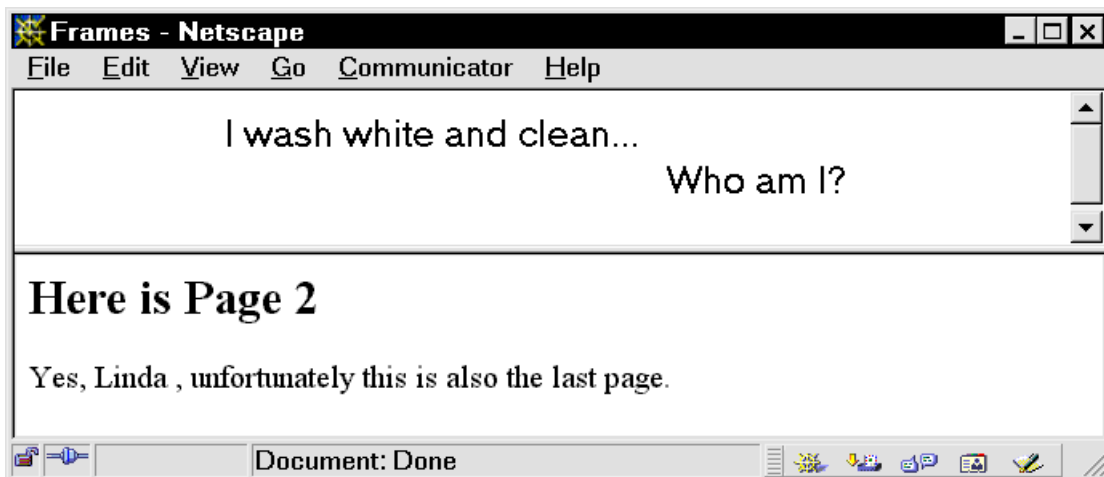
With `document.location.href`, the address of the page to be displayed is given. In our case, this is the HTML file **PAGE2.HTM**.

This file looks like this:

```
<html>
<head>
</head>
<body>
<h2>Here is Page 2</h2>
Yes,
<script>
document.write(parent.name)
</script>
, unfortunately this is also the last page.
</body>
</html>
```

The user's name that is saved in the frameset is displayed. Thus we have accomplished the task of giving a variable a value in one HTML file and reading it out in another one.

And this even works – the browser produces the following:


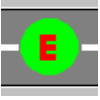
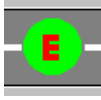
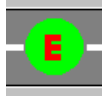




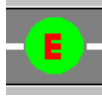




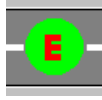


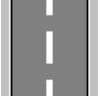

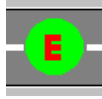



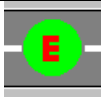
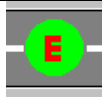



Quiz

Now I'd like to show you a longer exemplary application, which will make use of most of the contents of the foregoing sections. We're going to make a quiz game where the surfer has to drive through a labyrinth with a car. There is a start field and a destination field; in between lay several event fields. If the car crosses an event field, a question is posed, which, if answered correctly, will earn the player 100 points. If the player answers the question incorrectly, 40 points will be deducted from their score. In addition, 25 points are deducted for gasoline for each move the player makes.

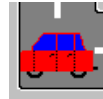
Note: if you're too lazy to type the entire lengthy source code in by yourself, or if you're artistically challenged and don't want to draw any pictures: all source code and graphics for this application can be downloaded at www.knowware.de/javascript. {MM: please specify the correct URL here!}

Here is the order of the pictures of the labyrinth and their names. All horizontal fields are event fields. The graphic design is up to you:

 START.GIF	 HORIZ.GIF (for horizontal)	 HORIZ.GIF (for horizontal)	 HORIZ.GIF (for horizontal)	 RT.GIF (for right top)
 DEST.GIF (for destination)	 EMPTY.GIF (empty field)	 LT.GIF (for left top)	 HORIZ.GIF (for horizontal)	 RB.GIF (for right bottom)
 LB.GIF (for left bottom)	 RT.GIF (for right top)	 RBR.GIF (for right branch)	 HORIZ.GIF (for horizontal)	 RT.GIF (for right top)
 EMPTY.GIF (empty field)	 VERT.GIF (for vertical)	 LB.GIF (for left bottom)	 HORIZ.GIF (for horizontal)	 LBR.GIF (for left branch)
 EMPTY.GIF (empty field)	 LB.GIF (for left bottom)	 HORIZ.GIF (for horizontal)	 HORIZ.GIF (for horizontal)	 RB.GIF (for right bottom)

For all files, with the exception of **EMPTY.GIF**, you must also create a version with a car. When you're doing this, add an **M** in front of the file name. For example, in addition to **LB.GIF**, there should be **MLB.GIF**:

The game should consist of a frameset. On the left side, you'll see the labyrinth; at the top right, the steering wheel of the car alternating with the questions; at the bottom right, the player's point total. Here is the frameset file, which is called



LABYRINTH.HTM:

```
<html>
<head>
<title>Quiz</title>
<script language="JavaScript">
<!--
var points=0
var x=1
var y=1
var answered=0

var afield=new Array
for (var i=1; i<=5; i++){
afield[i]=new Array
}
afield[1][1]="start"
afield[2][1]="horiz"
afield[3][1]="horiz"
afield[4][1]="horiz"
afield[5][1]="rt"
afield[1][2]="destination"
afield[2][2]="empty"
afield[3][2]="lt"
afield[4][2]="horiz"
afield[5][2]="rb"
afield[1][3]="lb"
afield[2][3]="rt"
afield[3][3]="rbr"
afield[4][3]="horiz"
afield[5][3]="rt"
afield[1][4]="empty"
afield[2][4]="vert"
afield[3][4]="lb"
afield[4][4]="horiz"
afield[5][4]="lbr"
afield[1][5]="empty"
afield[2][5]="lb"
afield[3][5]="horiz"
afield[4][5]="horiz"
afield[5][5]="rb"

var question=new Array
for (var i=1; i<=40; i++){
question[i]=new Object
question[i].asked=false
}
question[1].text="On which planet are there clouds made of
sulphuric acid?"
question[1].answer1="Venus"
question[1].answer2="Mercury"
```

```

question[1].answer3="Mars"
question[1].answer=1

question[2].text=" Which gas makes it possible for a match to
burn?"
question[2].answer1="nitrogen"
question[2].answer2="hydrogen"
question[2].answer3="oxygen"
question[2].answer=3

question[3].text="What do monks do in a refectory?"
question[3].answer1="pray"
question[3].answer2="work"
question[3].answer3="eat"
question[3].answer=3

question[4].text="Of which country is Newfoundland a part?"
question[4].answer1="Denmark"
question[4].answer2="Canada"
question[4].answer3="USA"
question[4].answer=2
...

// -->
</script>
</head>
<frameset cols="2*,*" border=3>
<frame src="laby.htm" marginwidth=10 marginheight=10>
<frameset rows="2*,*" border=0 noresize>
<frame src="steering.htm" marginwidth=10 marginheight=10>
<frame src="points.htm" marginwidth=10 marginheight=10>
</frameset>
</frameset>
</html>

```

This file is long, but it's easy to understand. At the beginning, all the variables are declared, since the main file never changes and the variables are therefore kept for the duration of the game.

`points` is supposed to keep track of the player's score and is assigned an initial value of 0. `x` and `y` specify the horizontal and vertical position of the car, which is started in the position 1/1.

The variable `answered` counts the questions already answered. We'll see later what this is good for. It too is assigned an initial value of 0.

The variable `afield` is declared as an array, while the variables `afield[1]` through `afield[5]` are designated as data fields using the `for` loop. There are, therefore, five arrays, each of which covers one line of the playing field.

Next, the arrays are assigned their corresponding picture files without the extension `.GIF`. For example, the position 1/2 is the destination, hence the line `afield[1][2]="destination"`. After this, the array `question` is created – I'm using 40 questions. `question[1]` through `question[40]` are declared as objects using a `for` loop (see the section on "User-Defined Objects" on page 46). The property `asked` is initially set to `false`, which means that the question hasn't been asked yet.

Next, the properties of all questions are determined. For reasons of space, I haven't shown you questions 5 to 40. `text` is the question itself, `answer1`, `answer2`, and `answer3` are the possible answers, and `answer` contains the index of the correct answer (for example, if

`answer1` contains the correct answer). Of course you must think up questions 5 to 40 yourself and set their properties analogously.

The next step is to create the simplest of the three frame files, `POINTS.HTM`. It outputs the value of the variable `points` for the frameset.

```
<html>
<head>
</head>
<body>
<center>
<h2>Your Score:</h2>
<font color="0000ff" size=6>
<b>
<script>
document.write(parent.points)
</script>
</b>
</font>
</center>
</body>
</html>
```

Even `LABY.HTM`, the actual labyrinth, is surprisingly easy:

```
<html>
<head>
<script language="JavaScript">
<!--
function print_laby(){
for (var i=1; i<=5; i++){
document.write("<tr>")
for (var j=1; j<=5; j++){
document.write('<td><img src=""')
if (parent.x==j & parent.y==i){
document.write("m")
}
document.write(parent.ffield[j][i] + '.gif"></td>')
}
document.write("</tr>")
}
}
//-->
</script>
</head>
<body bgcolor="000000">
<center>
<table border=0 cellpadding=0 cellspacing=0>
<script>
print_laby()
</script>
</table>
```

```

</center>
</body>
</html>

```

In the body of the HTML file, a table is opened, then the function `Funktion print_laby` is called. Finally the table is closed again.

The function consists mainly of two `for` loops. The first one, with the counter variable `i`, counts the rows; the second one, with the counter variable `j`, counts the columns. At the beginning of each iteration of the first loop, a column is opened with `document.write("<tr>")`. After that, a cell with a picture is opened five times. Using an `if` statement, the program checks to see whether the counter variables match the coordinates of the car – if so, then an `m` is added to the HTML file name for the changed graphic. Next, `parent.ffield[j][i]` writes the file name of the corresponding picture, which is completed with the file extension `.gif` and the cell is closed.

The most complicated file is `STEERING.HTM`:

```

<html>
<head>
<script language="JavaScript">
<!--
function print_steering() {
document.write("<tr><td></td><td>")
var f=parent.ffield[parent.x][parent.y]
if (f=="vert" || f=="lb" || f=="rb" || f=="rbr" || f=="lbr"){
document.write('<a href="javascript:go(0,-1)"></a>')
}
else {
document.write('')
}
document.write("</td><td></td></tr><tr><td>")
if (f=="horiz" || f=="rb" || f=="rt" || f=="lbr"){
document.write('<a href="javascript:go(-1,0)"></a>')
}
else {
document.write('')
}
document.write('</td><td></td><td>')
if (f=="horiz" || f=="lb" || f=="lt" || f=="rbr" || f=="start"){
document.write('<a href="javascript:go(1,0)"></a>')
}
else {
document.write('')
}
document.write("</td></tr><tr><td></td><td>")
if (f=="vert" || f=="lb" || f=="rt" || f=="rbr" || f=="lbr"){
document.write('<a href="javascript:go(0,1)"></a>')
}
}






```

```

else {
document.write('')
}
document.write("</td><td></td></tr>")
}
function go(x,y){
parent.points=parent.points-25
parent.x=parent.x + x
parent.y=parent.y + y
parent.frames[0].location.href="laby.htm"
if (parent.ffield[parent.x][parent.y]=="horiz"){
parent.frames[1].location.href="questions.htm"
}
else if (parent.ffield[parent.x][parent.y]=="destination"){
parent.frames[1].location.href="destination.htm"
}
else{
parent.frames[1].location.href="steering.htm"
}
parent.frames[2].location.href="points.htm"
}
//-->
</script>
</head>
<body>
&nbsp;  <p><center>
<table border=0 cellpadding=0 cellspacing=0>
<script>
print_steering()
</script>
</table>
</center>
</body>
</html>

```

In the body, after some spaces intended to create a vertical-centering effect, a table is opened. This time the function `print_steering` is called and then the table is closed again. In order to understand this function, I'll show you how the steering is constructed, namely with a table:

	 UPRED.GIF	
 LEFTRED.GIF	 STEERING.GIF	 RIGHTRED.GIF
	 DOWNRED.GIF	

The four arrows, this time in gray, must be stored with similarly-manipulated names (for example, `LEFTGRAY.GIF`). These files too are available to you at www.knowware.de/javascript. {MM: Please check this URL!} `print_steering` first issues the HTML commands for the empty cell in the upper left, then a cell is opened. The file name of the picture in the car's current position is read out of `parent.ffield[parent.x][parent.y]` into the variable `f`. Using `if` statements, the

program checks to see whether the car is on a field from which it is allowed to drive upwards. `||` means “or” – you must draw two vertical lines, which you can create using `SHIFT+\` (hold the `SHIFT` key down, then press the backslash key) on your keyboard. If the car is allowed to drive upwards, a reference that points to the function `go(0,-1)` displays the red arrow pointing upwards. If the car is not allowed to drive upwards (`else` = otherwise, in conjunction with `if`), then since there is no reference, the gray graphic is displayed. The function `go` will be described below; the first parameter determines the x movement (given vertical motion, this value is 0), the second parameter determines the y movement (-1 since the car is moving upwards).

The table is completed analogously; in the middle, the graphic `STEERING.GIF` is displayed. The function `go` deducts 25 points from the player’s score for gasoline. Then the file `LABY.HTM` is displayed in the left frame – but since the coordinates have changed, the car will be put in a new position. Finally, the program checks to see whether the car is on a horizontal field, that is, a question field. If so, the file `questions.htm` is displayed in the upper right frame. If, by contrast, the car is on the destination field, then the file `destination.htm` is displayed in this frame. Otherwise, the file `steering.htm` is updated, since eventually other arrows must be displayed. Last but not least, the user’s score is updated.

`questions.htm` is a much less complicated file:

```
<html>
<head>
<script language="JavaScript">
<!--
function question() {
if (parent.answered>39) {
document.write('You have already answered all the questions.
<p><center><a href="steering.htm">Continue</a></center>')
}
else{
a=Math.round(Math.random()*39)+1
while (parent.question[a].asked==true) {
a=Math.round(Math.random()*39)+1
}
parent.answered++
parent.question[a].asked=true
document.write(parent.question[a].text + "<p>")
document.write('<a href="javascript:answer(a,1)">' +
parent.question[a].answer1 + '</a><br>')
document.write('<a href="javascript:answer(a,2)">' +
parent.question[a].answer2 + '</a><br>')
document.write('<a href="javascript:answer(a,3)">' +
parent.question[a].answer3 + '</a><br>')
}
}

function answer(index,tip) {
if (parent.question[index].answer==tip) {
alert("Correct! You get 100 Points!")
parent.points=parent.points+100
}
```

```

    }
    else{
    alert("Sorry, that is wrong. You lose 40 points.")
    parent.points=parent.points-40
    }
    parent.frames[1].location.href="steering.htm"
    parent.frames[2].location.href="points.htm"
  }

  //-->
</script>

</head>
<body>
<script>
question()
</script>
</body>
</html>

```

This time, the function `question()` is called in the body.

Next, the program checks to see whether the number of questions that the player has already answered equals the number of existing questions. If that's the case, then the player is informed of this and the file `STEERING.HTM` is referenced.

Using `a=Math.round(Math.random()*39)+1`, a random number between 1 and 40 is selected. `Math.random()` selects a random decimal part between 0 and 1, while `Math.random()*39` selects a random number between 0 and 39. `Math.round` rounds the result to a whole number. In order to get a number between 1 and 40, you must add 1. These actions will be repeated in the `while` loop until the program finds a question that it has not yet posed. The number of the already-posed questions is increased by 1 and the property `asked` for the corresponding question is set to `true`.

Next, the text of the question, followed by a blank line, is displayed.

Finally, the three possible answers are displayed as links, which reference the function `answer` with the parameters `a` (index of the question) and the number of the selected answer.

The function `answer` checks to see whether the selected answer corresponds to the correct answer. If so, then a message is displayed and 100 points are added to the player's score; if not, then a message is displayed and 40 points are deducted from the player's score. Finally, in the upper right frame, the steering wheel is displayed and the player's score is updated in the lower right frame.

The last file that you'll need is called `DESTINATION.HTM`:

```

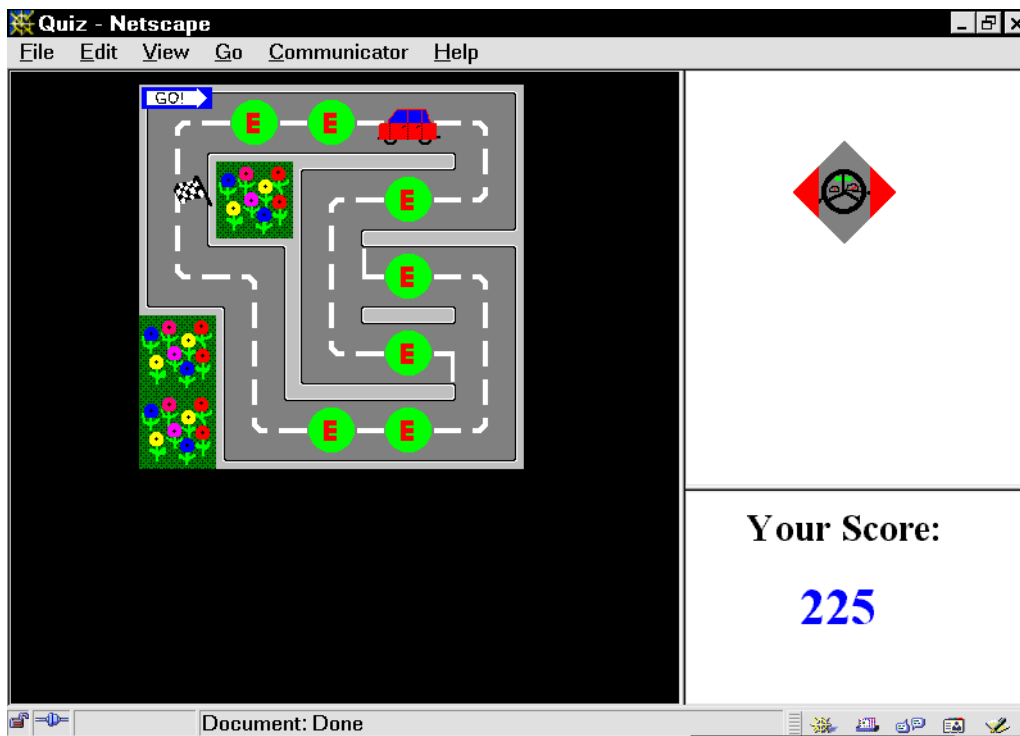
<html>
<head>
</head>
<body>
<center>
<h2>Congratulations!</h2>
<b>You have arrived at the destination with

```

```
<script>
document.write (parent.points)
</script>
  points.<p>
<a href="labyrinth.htm" target="_parent">New Game</a>
</b>
</center>
</body>
</html>
```

This file shows the successful player how many points they had at the destination and offers them the opportunity to play again.

Here's a picture of the browser during the game:



You can try out the whole game at www.knowwareglobal.com/javascript/

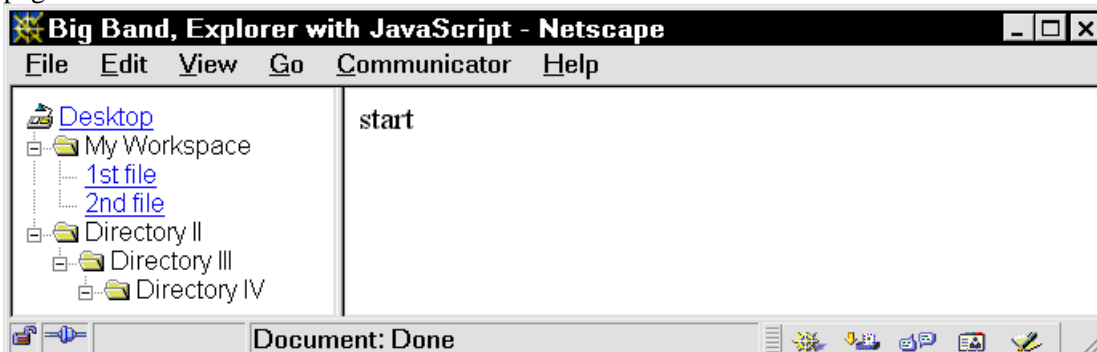
The Explorer

The Project

After the quiz game, which is an application for the entertainment of the surfer, I'd like to present a serious application at the end of this booklet. It should solve a problem common to Web sites with a large number of pages; it becomes impossible to get an overview of what is where. Our goal is to create buttons with names like "Continue" or "Back," since after you have clicked back and forth on a site a few times, you've probably lost the overview completely.

Often frames are used to solve this problem: on the left-hand side of the browser window, a frame displays links to various pages, while the right-hand frame displays the page that's been selected. Frames work pretty well -- as long as the number of pages remains in the realm of one or low two digits.

If there are too many pages on a site, however, you can help clear up the confusion in another way. To do this, you'll also use two frames: the right-hand frame will again show the contents of the page you've selected, but the left-hand frame will display a tree structure like the Windows Explorer does. By opening and closing folders, you'll see only the pages relevant to your direct calls. To give you a better picture of the project, here's a screen shot of the finished page.



The Practice

Since a full-blown practical example would require many, many directories and files, here I'll just show you the basic structure of an Explorer system. To customize this project for your Web site, see the section entitled "Customization" below.

The Main Page

The main page is, in principle, a completely normal frame page, but it must contain a JavaScript portion. This is explained below.

```
<html>
<head>
<title>Big Band, Explorer with JavaScript</title>
<script language="JavaScript">
<!--
function init_ex(level,opened,description,filename) {
  this.level=level
  this.opened=opened
  this.description=description
  this.filename=filename
}
var ex=new Array
```

```

ex[0]=new init_ex(0,true,"Desktop","start.htm")
ex[1]=new init_ex(1,true,"My Workspace","")
ex[2]=new init_ex(2,true,"1st file","xy.htm")
ex[3]=new init_ex(2,true,"2nd file","xyz.htm")
ex[4]=new init_ex(1,true,"Directory II","")
ex[5]=new init_ex(2,true,"Directory III","")
ex[6]=new init_ex(3,true,"Directory IV","")
ex[7]=new init_ex(-1,true,"","")
//-->
</script>
<frameset cols="210,*">
<frame src="explorer.htm">
<frame src="start.htm" name="contents">
</frameset>
</head>
</html>

```

The workings of the function `init_ex` were described in the section “User-Defined Objects.” If you don’t remember that section anymore, don’t worry; the following sections will make the workings of this function clear to you all over again.

The line `var ex=new Array` creates a data field. The variables `ex[1]`, `ex[2]`, `ex[3]`, etc. are defined all at one time. Each `ex[x]` is supposed to be responsible for a line in the Explorer, although not all the lines must be displayed at once. A line contains several pieces of information: is the line a folder or a file? If the line is a folder, is it open? If the line is a file, what address has been assigned to it? Which text should be displayed in the line in the Explorer?

All of this information is written into the variable `ex[x]` using the function `init_ex`, as follows:

```
ex[x]=new init_ex(Level,true/false,"Displaytext","Filename")
```

Now you’ll need some explanation of the arguments:

The first argument, “level,” determines how far from the main directory the folder or file in question is located. The main folder, that is, “Big Band,” is the only line on level 0. Folders and files that are located directly underneath it, for example, “Winds,” are on level 1. Files and folders below this are on level 2, etc.

If a line is concerned with a folder, the next argument is relevant; it determines whether the folder is open (true) or closed (false) by default. For files, this argument is meaningless, but it must still be set to true or false so that the function is passed the right number of parameters. The next argument determines the text that the Explorer will display on the corresponding line. This text should be written in accordance with the HTML standard, that is, you should write „ä“ for „ä“.

Finally, the last argument must remain empty for folders; for files, you must specify the appropriate file name to which the line refers.

After the last line of the Explorer, you must still assign a last `ex[x]`. Its values are unimportant; what’s important is that it will serve as Level –1. This line with Level –1 serves as a marker stating that all lines have been set.

After these assignments, you can call up the level of a line using `ex[x].level`. You can use `ex[x].opened` to determine whether a folder is open (if the line in question concerns a folder). You can use `ex[x].description` to see the text that will be displayed in the Explorer, and, last but not least, `ex[x].filename` will show you the file name of the file in question (provided the line concerns a file).

Don’t forget to give the right-hand frame a name (“contents”); this will be important later on.

Finally, save the page as **INDEX.HTM**.

The Content Page

In the right frame, the contents that the user has chosen are displayed. This means that for every page you can call up, you need to create an HTML file. In our case, we'll need a start page **START.HTM**, a page called **XY.HTM**, and a page called **XYZ.HTM**. Here's an example of how **START.HTM** might look:

```
<html>
<head>
</head>
<body>
start
</body>
</html>
```

Since the page will be displayed in a frame, it doesn't need a title of its own. Naturally you can also add graphics, tables, and other elements.

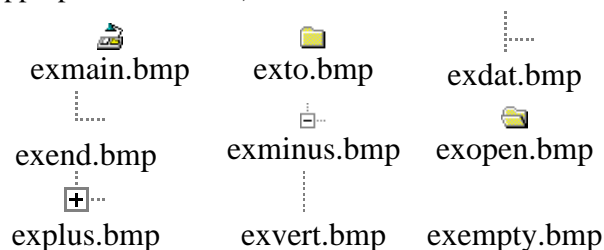
The Explorer Page

In principle, the Explorer page must "create itself" like the page with the labyrinth in the last section. Information about individual lines must be read out from the main page and then translated into HTML code. The basic structure should be familiar to you from the quiz game; it should look like this (the meaning of the **max** variable will be explained later on):

```
<html>
<head>
<script language="JavaScript">
<!--
var max=1000
...
//-->
</script>
</head>
<body alink="FF0000" vlink="0000FF">
<script>
print_explorer()
</script>
</body>
</html>
```

The function **print_explorer** takes over the construction of the whole page.

In order to imitate the graphical structure of the Windows Explorer, you'll need the following graphics (each with an appropriate file name). You can also download them from the website.



In order to guarantee that the browser will display these graphics right next to one another without a lot of space in between them, you'll need to create a separate table for each line. At first glance this may seem really roundabout, but I don't know of any other solution that is Netscape- and Microsoft- compatible. The following function, which is called up when the

page is being built, must be declared in the JavaScript portion of the file.

```
function print_explorer() {
  var i=0
  while (parent.ex[i].level!=-1) {
    if (parent.ex[i].level<=max) {
      document.write("<table border=0 cellpadding=0 cellspacing=0>")
      if (parent.ex[i].level==0) {
        exmain()
      }
      else if (parent.ex[i].filename=="") {
        document.write("<tr>")
        exempty(i)
        exfolder(i)
        document.write("</tr>")
      }
      else {
        document.write("<tr>")
        exempty(i)
        exfile(i)
        document.write("</tr>")
      }
      document.write("</table>")
    }
    i++
  }
}
```

With `var i=0`, a counter variable is declared that will count the individual lines of the Explorer.

With `while (parent.ex[i].level!=-1)`, the loop will run until the last line is in place. This is also why the line that sets the level to -1 is necessary.

Next, the program checks to see whether the level is smaller than or equal to the contents of the variable `max`.

If a folder is closed, then later on the variable `max` will be set to the level of the folder. This prevents the display of subordinate files and folders (which have a higher level).

If a line should be displayed, then a table will be opened.

Next, the program checks to see what kind of line the line in question is: if it's level 0, then it represents the main folder, so the function `exmain` is called.

If, by contrast, the line concerns a regular folder, then the associated file name is empty, and the function `exempty` is called. This function is responsible for displaying empty fields in front of the folder so that it will appear far enough to the right. It also draws vertical lines that lead to other files or folders. After this, the function `exfolder` is called; it draws the folder. The argument `i` specifies the line number for the function. The empty fields, vertical lines, and the folder are placed in a table column, which is the only one in the table, and has already been opened and closed in the function `print_explorer`.

In all other cases, the line concerns a file. The function calls are analogous to the lines for folders described above, but instead of `exfolder`, you'll need to call `exfile`.

Finally, the table is closed and the counter variable is incremented so that the program can continue with the next line.

Now we'll discuss the individual functions, beginning with `exmain`:

```
function exmain() {
  document.write("<tr><td><a href='javascript:unopen(0) '><img
src='exmain.gif' border=0></a></td><td><nobr><font face='Arial'
size=2>&nbsp;<a href='\" + parent.ex[0].filename + \"'
```

```

target='contents'>" + parent.ex[0].description +
"</a></font></noabr></td></tr>")

if (parent.ex[0].opened===false) {
max=0
}
else{
max=1000
}
}

```

This function is only called for the first line. It draws the icon for the main folder in a cell of the table and then refers the program to the function `unopen`, which is responsible for the opening and closing of folders and will be discussed later on.

Next, the name of the main folder is written into a new cell (`parent.ex[0].description`), which refers to the main page in the right-hand frame (`parent.ex[0].filename`). The font tags serve to produce a typeface that is appropriate for the Explorer. Using an `if` command, the program determines whether the main folder is open or closed. In the latter case, the variable `max` is set to 0 so that no subfolders or files will be displayed; otherwise, `max` will be set to a high enough value to enable the display of all files and folders. Next, we'll talk about

`exfolder`:

```

function exfolder(i) {
var picture1="exopen.gif"
var picture2="exminus.gif"
if (parent.ex[i].opened===false) {
picture1="exto.gif"
picture2="explus.gif"
}
document.write("<td><a href='javascript:unopen(" + i + ")'><img
src='" + picture2 + "' border=0></a></td><td><a
href='javascript:unopen(" + i + ")'><img src='" + picture1 + "'
border=0></a></td><td><noabr><font face='Arial' size=2>&nbsp;&nbsp;&nbsp;"+
parent.ex[i].description + "</font></noabr></td>")
if (parent.ex[i].opened===false) {
max=parent.ex[i].level
}
else{
max=1000
}
}
}

```

In the first lines, the variables `picture1` and `picture2` are assigned the file names of the appropriate graphics. If a folder is open, then the program should display a minus sign and an open folder; if a folder is closed, then it should show a plus sign and a closed folder.

The further course of this function corresponds to that of `exmain`. The only difference is that instead of the index 0 for the main directory, an `i` for the current line should be used. Also, two pictures will be displayed; the names of the files are taken from the variables `picture1` and `picture2`. There is no reference to a page since we're concerned with a folder here.

The third function for displaying individual lines is called `exfile`:

```

function exfile(i) {
var picture="exdat.gif"
if (parent.ex[i].level!=parent.ex[i+1].level) {
picture="exend.gif"
}
document.write("<td><img src='" + picture +
"'></td><td><noabr><font face='Arial' size=2>&nbsp;&nbsp;&nbsp;<a href='" +

```

```

parent.ex[i].filename +' target='contents'>" +
parent.ex[i].description + "</a></font></nobr></td>")
}

```

Next, the program determines whether the file follows a folder or file with the same level. If so, the variable `picture` is assigned the file name EXDAT.GIF; if not, it's assigned EXEND.GIF. The line is completed in the same way as in the function `exmain`. Note that it is not necessary to change the variable `max` since a file cannot be open or closed like a folder.

The function that draws the leading empty cells has not yet been mentioned; here it is:

```

function exempty(i) {
for (j=0; j<=parent.ex[i].level-2; j++){
paint=false
k=i+1
while ((parent.ex[k].level>=j-2) && (parent.ex[k].level!==-1)) {
if (parent.ex[k].level==j+1) {
paint=true
}
k++
}
if (paint==true) {
document.write("<td><img src='exvert.gif'></td>")
}
else {
document.write("<td><img src='exempty.gif'></td>")
}
}
}

```

Using a `for` loop, all possible places for empty spaces or vertical lines are counted through using the counter variable `j`. The `while` loop, using the counter variable `k`, checks all previous lines until it encounters a line with a higher level, or the end of the file or folder with the same level as the current line. If this is the case, the variable `paint` is set to `true`. Finally, the program will draw a vertical line if there is another line with the same level; otherwise, it will add an empty space.

The simplest function is the function `unopen`:

```

function unopen(nr) {
parent.ex[nr].opened=!parent.ex[nr].opened
parent.frames[0].location.href="explorer.htm"
}

```

It inverts the variable that determines whether a folder is open and loads the Explorer page into the frame anew.

For the sake of clarity, here's the entire source code:

```

<html>
<head>
<script language="JavaScript">
<!--
var max=1000
function print_explorer() {
var i=0
while (parent.ex[i].level!=-1) {
if (parent.ex[i].level<=max) {
document.write("<table border=0 cellpadding=0 cellspacing=0>")
if (parent.ex[i].level==0) {
exmain()
}
else if (parent.ex[i].filename=="") {

```

```

document.write("<tr>")
exempty(i)
exfolder(i)
document.write("</tr>")
}
else{
document.write("<tr>")
exempty(i)
exfile(i)
document.write("</tr>")
}
document.write("</table>")
}
i++
}
}
function exmain() {
document.write("<tr><td><a href='javascript:unopen(0) '><img
src='exmain.gif' border=0></a></td><td><no><font face='Arial'
size=2>&nbsp;<a href='\" + parent.ex[0].filename + \"'
target='contents'>\" + parent.ex[0].description +
\"</a></font></no></td></tr>")

if (parent.ex[0].opened===false) {
max=0
}
else{
max=1000
}
}
function exfolder(i) {
var picture1="exopen.gif"
var picture2="exminus.gif"
if (parent.ex[i].opened===false) {
picture1="exto.gif"
picture2="explus.gif"
}
document.write("<td><a href='javascript:unopen(\" + i + \") '><img
src='\" + picture2 + \"' border=0></a></td><td><a
href='javascript:unopen(\" + i + \") '><img src='\" + picture1 + \"'
border=0></a></td><td><no><font face='Arial' size=2>&nbsp;<\" +
parent.ex[i].description + \"</font></no></td>")
if (parent.ex[i].opened===false) {
max=parent.ex[i].level
}
else{
max=1000
}
}
function exfile(i) {
var picture="exdat.gif"
if (parent.ex[i].level!=parent.ex[i+1].level) {
picture="exend.gif"
}
document.write("<td><img src='\" + picture +
\"'></td><td><no><font face='Arial' size=2>&nbsp;<a href='\" +

```

```

parent.ex[i].filename +'' target='contents'>" +
parent.ex[i].description + "</a></font></nobr></td>")
}
function exempty(i){
for (j=0; j<=parent.ex[i].level-2; j++){
paint=false
k=i+1
while ((parent.ex[k].level>=j-2) && (parent.ex[k].level!==-1)){
if (parent.ex[k].level==j+1){
paint=true
}
k++
}
if (paint==true){
document.write("<td><img src='exvert.gif'></td>")
}
else{
document.write("<td><img src='exempty.gif'></td>")
}
}
}
function unopen(nr) {
parent.ex[nr].opened=!parent.ex[nr].opened
parent.frames[0].location.href="explorer.htm"
}
//-->
</script>
</head>
<body alink="FF0000" vlink="0000FF">
<script>
print_explorer()
</script>
</body>
</html>

```

The complete file must be stored as EXPLORER.HTM.

Customization

Of course it could happen that you haven't understood all the steps necessary for creating the pages of this quite complex project down to the last detail. Therefore, I would like to show you here how you can customize the Explorer to suit your own page.

The only place where you'll need to change anything is in the file INDEX.HTM. There you must be careful that the list with `ex[i]` is numbered continuously. With the first argument, the level, you can determine how far to the right the file or folder will be placed. In the case of folders, the second argument determines whether the folder should be open by default (true) or closed (false). After that comes a description of the line. Finally, there is the argument with the associated file name, which, in the case of folders, must be empty. Don't forget the last line with the level -1. And be sure that you only give four arguments for each line.

Reserved Words

The following words are reserved and cannot be used as function or variable names:

<code>abstract</code>	<code>instanceof</code>
<code>boolean</code>	<code>int</code>
<code>break</code>	<code>interface</code>
<code>byte</code>	<code>long</code>
<code>case</code>	<code>native</code>
<code>catch</code>	<code>new</code>
<code>char</code>	<code>null</code>
<code>class</code>	<code>package</code>
<code>const</code>	<code>private</code>
<code>continue</code>	<code>protected</code>
<code>debugger</code>	<code>public</code>
<code>default</code>	<code>return</code>
<code>delete</code>	<code>short</code>
<code>do</code>	<code>static</code>
<code>double</code>	<code>super</code>
<code>else</code>	<code>switch</code>
<code>enum</code>	<code>synchronized</code>
<code>export</code>	<code>this</code>
<code>extends</code>	<code>throw</code>
<code>false</code>	<code>throws</code>
<code>final</code>	<code>transient</code>
<code>finally</code>	<code>true</code>
<code>float</code>	<code>try</code>
<code>for</code>	<code>typeof</code>
<code>function</code>	<code>var</code>
<code>goto</code>	<code>void</code>
<code>if</code>	<code>volatile</code>
<code>implements</code>	<code>while</code>
<code>import</code>	<code>with</code>
<code>in</code>	

When you're choosing variable and function names, you should be sure that they don't contain any empty spaces or strange characters (only `_` is allowed). Furthermore, you should not create names that are longer than 32 characters. Despite the freedom you have in naming, I would strongly recommend that you name your variables so that you'll still know what their function is four weeks down the road. So, for example, you shouldn't name your variables `a`, `b`, `c`, `d`, `e` and `f`; it would be more sensible to call them `input`, `result`, `counter`, `day`, `month`, and `year` instead.

The Last Word...

I hope that in addition to learning something, you've had some fun reading this booklet. I am very happy to get your feedback, regardless of whether it's positive or negative. Please e-mail me at martin.baier@gmx.net. As I've already mentioned, you'll find additions to this booklet, all source code, and graphics at {MM: please fill in the correct URL!}. There you'll also find references to sites that will help you improve your knowledge of JavaScript. I wish you much fun and success programming your own applications – please e-mail me the URL when you've published them!

Martin Baier

Array, 46
Background color, 29
Calculator, 42
checked, 38
Color, 29
conditional, 27
Date, 11
defaultChecked, 38
defaultValue, 38
document, 29
document.forms, 38
document.frames, 35
document.location, 31
document.referrer, 31
document.title, 31
document.write, 11
elements, 38
else, 53
Event-Handler, 13
Explorer, 58
focus, 38
for, 23
Frames, 35; 46
Function, 19
getHours, 11
getMinutes, 11
Header, 7
Hello World, 9
if, 27
Image, 32
Incorporation, 7
 Body, 8
javascript, 18
length, 38; 44
Link, 12
Loops, 23
Math, 22
Moving Text, 44
Objects, 46
onBlur, 16
onChange, 16
onClick, 17
onFocus, 14
onLoad, 13
onMouseOut, 12; 14
onMouseOver, 14
onSubmit, 18
onUnload, 13
Operation, 22
Parameters, 10; 19
Pictures, 32
Quiz, 49
Reference, 12
Repetition, 23
round, 22
select, 38
selectedIndex, 38
setTimeout, 45
src, 32
Status line, 12; 45
Strings, 44
substring, 44
Text color, 29
this, 46
Time, 11
toLowerCase, 44
toUpperCase, 44
value, 38
Variable, 20
 global, 21
 local, 20
while, 25
window, 29