Introduction to JAVASCRIPT

What is JavaScript

JavaScript is a powerful, object-based scripting language which is being developed by Netscape. It can be embedded directly into HTML pages. It allows to create dynamic, interactive Web-based applications that run completely within a Web browser.

JavaScript is the language of choice for developing Dynamic HTML (DHTML) content. JavaScript can be integrated effectively with CGI and JAVA, to produce sophisticated Web applications. In many cases JavaScript eliminates the need for complex CGI scripts and Java applets altogether.

The scripting language is:

- A scripting language is one in which people think they're writing scripts, and a programming language is one in which they think they're writing programs. You can come up with rules of thumb, e.g. scripting languages tend to be interpreted, but it's easy to find exceptions to any of them.

Nevertheless, there are also different views to a scripting language, described by the properties and the role of language in programming process:

- Scripting languages often follow the syntax and semantics of command languages. For instance, many scripting languages do not require quoting of string literals, but rather require explicit evaluation of variables (*x* denotes the string "*x*", but *$x* denotes the value of the variable *x*);
- Scripting languages make it easy to call system commands, prepare their arguments, and manipulate their results. They generally have some built-in primitives for manipulating file and directory names, argument lists, environment variables, etc.;
- Scripting languages generally are good at handling strings, and don't emphasize numerical manipulation;
- Since calling system commands is generally much more expensive than script execution itself, there is little emphasis on run-time efficiency, therefore they are often implemented using interpreters, bytecode interpreters, or macro processors.

Client side and server side:

JavaScript is not Java!

Many people believe that JavaScript is the same as Java because of the similar names. This is **not** true though. It would go too far at the moment to show all the differences - so just memorize that JavaScript is *not* Java. For further information on this topic please read the introduction provided by Netscape or in the book of Stefan Koch: *http://rummelplatz.uni-mannheim.de/~skoch/js/*

Running JavaScript

What is needed in order to run scripts written in JavaScript? You need a JavaScript-enabled browser - for example the Netscape Navigator (since version 2.0) or the Microsoft Internet Explorer (MSIE - since version 3.0). Since these two browsers are widely spread many people are able to run scripts written in JavaScript. This is certainly an important point for choosing JavaScript to enhance your web-pages.

Embedding JavaScript into a HTML-page

JavaScript code is embedded directly into the HTML-page, it's illustrated by an easy example:

```
<html>
<body>
<br>
This is a normal HTML document.
<br>
  <script language="JavaScript">
    document.write("This is JavaScript!")
  </script>
<br>
Back in HTML again.
</body>
</html>
```

This looks like a normal HTML-file, and the only new thing is the part:

```
  <script language="JavaScript">
    document.write("This is JavaScript!")
  </script>
```

This is JavaScript. In order to see this script working save this code as a normal HTML-file and load it into your JavaScript-enabled browser. Here is the output generated by the file (if you are using a JavaScript browser you will see 3 lines of output):

This is a normal HTML document.
This is JavaScript!
Back in HTML again.

Everything between the <*script*> and the <*/script*> tag is interpreted as JavaScript code. There you see the use of

document.write()

- one of the most important commands in JavaScript programming.

document.write() is used in order to write something to the actual document (in this case this is the HTML-document). So this little JavaScript program writes the text *This is JavaScript!* to the HTML-document.

Non-JavaScript browsers

If the browser does not understand JavaScript, it does not know the <*script*> tag. It ignores the tag and outputs all following code as if it was normal text, so the user will see the JavaScript-code of our program inside the HTML-document.

There is a way for hiding the source code from older browsers. We will use the HTML-comments <*!*-- -->, and the new source code looks like this:

```
<html>
<body>
<br>
This is a normal HTML document.
<br>
  <script language="JavaScript">
  <!-- hide from old browsers
    document.write("This is JavaScript!")
  // -->
  </script>
<br>
Back in HTML again.
</body>
</html>
```

The output in a non-JavaScript browser will then look like this:

This is a normal HTML document.
Back in HTML again.

Without the HTML-comment the output of the script in a non-JavaScript browser would be:

This is a normal HTML document.
document.write("This is JavaScript!")
Back in HTML again.

Please note that you cannot hide the JavaScript source code completely. What we do here is to prevent the output of the code in old browsers - but the user can see the code through "*View document source"* nevertheless. There is no way to hinder someone from viewing your source code (in order to see how a certain effect is done).

Versions of JavaScript

The following table specifies what versions of client-side JavaScript are supported by various versions of Netscape Navigator and Microsoft Internet Explorer:

Version	**Netscape Navigator**	**Internet** Explorer
2	JavaScript 1.0	
3	JavaScript 1.1	JavaScript 1.0
4	JavaScript 1.2; not fully ECMA-262 compliant prior to version 4.5	JavaScript 1.2; EMCA-262 compliant

JavaScript Syntax

.IavaScript syntax is modeled on Java syntax, Java syntax, in turn, is modeled on C and C++ syntax. Therefore, C, C++, and Java programmers should find that Java Script syntax is comfortably familiar.

Case sensitivity

- JavaScript is a case-sensitive language. All keywords are in lowercase. All variables, function names, and other identifiers must be typed with a consistent capitalization.

Whitespace

- JavaScript ignores whitespace between tokens. Spaces, tabs, and newlines may be used to format and indent the code in a readable fashion.

Semicolons

- JavaScript statements are terminated by semicolons. When a statement is followed by a newline, however, the terminating semicolon may be omitted. Note that this places a restriction on where you may legally break lines in your JavaScript program: you may not break a statement across two lines if the first line can be a complete legal statement on its own.

Comments

- JavaScript supports both C and C++ comments. Any amount of text, on one or more lines, between /* and */ is a comment, and is ignored by JavaScript. Also, any text between // and the end of the current line is a comment, and is ignored. Examples:

```
// this is a single-line, C++-style comment.
/*
 * this is a multi-line, C-style comment.
 * here is the second line.
 */
/* another comment. */  // this too.
```

Identifiers

- variable, function, and label names are JavaScript identifiers. Identifiers are composed of any number of ASCII letters and digits, and the underscore (_) and dollar sign ($) characters. The first character of an identifier must not be a digit, however, and the $ character is not allowed in identifiers in JavaScript 1.0.

Keywords

- the following keywords are part of the JavaScript language, and have special meaning to the JavaScript interpreter. Therefore, they may not be used as identifiers:

break	**for**	**this**

case	**function**	**true**
continue	**if**	**typeof**
default	**import**	**var**
delete	**in**	**void**
do	**new**	**while**
else	**null**	**with**
export	**return**	
false	**switch**	

In addition, JavaScript reserves the following words for possible future extensions. You may not use any of these words as identifiers either:

catch	**enum**	**super**
class	**extends**	**throw**
const	**finally**	**try**
debugger		

Variables

- variables are declared, and optionally initialized, with the *var* statement:

```
var i;
var j = 1+2+3;
var k,l,m,n;
var x = 3, message = 'hello world';
```

Variable declarations in top-level JavaScript code may be omitted, but they are required to declare local variables within the body of a function.

JavaScript variables are *untyped:* - they can contain values of any data type.

Global variables in JavaScript are implemented as properties of a special global object. Local variables within functions are implemented as properties of the **`Argument`** object for that function.

Data Types

- JavaScript supports three primitive data types: *numbers*, *boolean values*, and *strings*. In addition, it supports two compound data types: *object* and *arrays*. Functions are also a first class data type in JavaScript, and JavaScript 1.2 adds support for regular expressions (described later) as a specialized type of object.

Numbers

- Numbers in JavaScript are represented in 64-bit floating-point format. JavaScript makes no distinction between integers and floating-point numbers. Numeric literals appear in JavaScript programs using the usual syntax: a sequence of digits, with an optional decimal point and an optional exponent. For example:

```
1
3.14
.0001
6.02e23
```

- integers may also appear in octal or hexadecimal notation. An octal literal begins with 0, and a hexadecimal literal begins with 0x:

`0377` the number 255 in octal
`0xFF` the number 255 in hexadecimal

- when a numeric operation overflows, it returns a special value that represents positive or negative infinity. When an operation underflows, it returns zero. When an operation such as taking the square root of a negative number yields an error or meaningless result, it returns the special value **`NaN`** , which represents a value that is *not-a-number*. Use the global function **`isNaN()`** to test for this value.

- The **`Number`** object defines useful numeric constants. The **`Math`** object defines various mathematical operations.

Booleans

- the boolean type his two possible values, rcpresented by the JavaScript keywords **`true`** and **`fa1se`**. These values represent *truth* or *falsehood, on* or *off, yes* or *no*, or anything else that can be represented with one bit of information.

Strings

- a JavaScript string is a sequence of arbitrary letters, digits, and other characters. The ECMA-262 standard requires JavaScript to support the full 16-bit Unicode character set. IE 4 supports Unicode, but Navigator 4 supports only the Latin-1 character set.

String literals appear in JavaScript programs between single or double quotes. One style of quotes may be nested within the other:

```
'testing'
"3. 14"
'name = "myform" '
"Wouldn't you prefer O'Reilly's book?"
```

When the backslash character **(\)** appears within a string literal, it changes or "escapes" the meaning of the character that follows it. The following table lists these special escape sequences:

Escape	Represents
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	'I'ab
\'	Apostrophe or single quote that does not terminate the string
\"	Double-quote that does not terminate the string
****	Single backslash character
\ddd	Character with Latin-1 encoding specified by three octal digits ddd
\xdd	Character with Latin-1 encoding specified by two hexadecimal digits dd
\udddd	Character with Unicode encoding specified by four hexaclecimal digits dddd
\n	n, where *n* is any character other than those shown above

The String class defines many methods that you can use to operate on strings. It also defines the **length** property, which specifies the number of characters in a string.

- The addition **(+)** operator concatenates strings.
- The equality **(=)** operator compares two strings to see if they contain exactly the same sequences of characters (this is compare-by-value, not compare-by-reference, as C, C++, or Java programmers might expect).
- The inequality operator **(!=)** does the reverse.
- The relational operators **(<, <=, >,** and **>=)** compare strings using alphabetical order.

JavaScript strings are *immutable*, which means that there is no way to change the contents of a string. Methods that operate on strings typically return a modified copy of the string.

Objects

- An *object* is a compound data type that contains any numer of properties. Each property has a name and a value. The **.** operator is used to access a named propertv of an object. For example, you can read and write property values of an object **o** as follows:

```
o.x = 1;
o.y = 2;
o.total = o.x + o.y;
```

Object properties are not defined in advance as they are in C, C++, or Java; any object can be assigned any property.

JavaScript objects are associative arrays: they associate arbitrary data values with arbitrary names. Because of this fact, object properties can also be accessed using array notation:

```
o["x"] = 1;
o["y"] = 2;
```

Objects are created with the **new** operator. You can create a new object with no properties as follows:

```
var o = new Object( );
```

Typically, however, you use predefined *constructors to* create objects that are members of a *class* of objects and have suitable properties and methods automatically defined. For example, you can create a **Date** object that represents the current time with:

```
var now = new Date( );
```

You can also define your own object classes and corresponding constructors.

In JavaScript 1.2, you can use object *literal* syntax to include objects literally in a program. An object literal is a comma-separated list of name/value pairs, contained within *curly* braces. For example:

```
var o = {x:l, y:2, total:31};
```

Arrays

- An array is a type of object that contains numbered values rather than named values. The **[ ]** operator is used to access the numbered values of an array:

```
a[0] = 1;
a[l] = a[0] + a[0];
```

The first element of a JavaScript array is element 0. Every array has a *length* property that specifies the number of elements in the array. The last element of an array is element *length - 1*.

You create an array with the **Array( )** constructor:

```
var a = new Array( );       // Empty array
var b = new Array(1O);      // 10 elements
var c = new Array(1,2,3);   // Elements 1,2,3
```

In.JavaScript 1.2, you can use array literal syntax to include arrays directly in a program. An *array* literal is a comma-separated list of values inclosed within square brackets. For example:

```
var a = [1,2,3];
var b = [1, true, [1,2], {x:l, y:2}, "Hello"];
```

The **Array** class defines a number of useful methods for working with arrays.

Functions and methods

- A function is a piece of JavaScript code that is defined once and can be executed multiple times by a program. A function definition looks like this:

```
function sum(x, y) {
  return x + y;
  }
```

Functions are invoked using the **()** operator and passing a list of argument values:

```
var total = sum(1,2);   // Total is now 3
```

In JavaScript 1.1, you can create functions using the **Function( )** constructor:

```
var sum = new Function ("x","y", "return x+y;");
```

In JavaScript 1.2, you can define functions using function literal syntax:

```
var sum = function(x,y) { return x+y; }
```

When a function is assigned to a property of an object, it is called a *method* of that object. Within the body of the function, the keyword **`this`** refers to the object for which the function is a property.

Within the body of a function, the **`arguments [ ]`** contains the complete set of arguments passed to the function. The **`Function`** and **`Arguments`** classes represent functions and their arguments,

null and undefined

- The JavaScript keyword **`null`** is a special value that indicates *"no value"*. If a variable contains **`null`**, you know, that it does not contain a valid value of any type.

There is one other special value in JavaScript: *the undefined value.* This is the value returned when you use an undeclared or uninitialized variable or when you use a non-existent object property. There is no JavaScript keyword for this value.

JavaScript hierarchy

- JavaScript organizes all elements on a web-page in a hierarchy.

Every element is seen as a object. Each object can have certain properties and methods. With the help of JavaScript you can easily manipulate the objects. For this it is very important to understand the hierarchy of HTML-objects. The following code is a simple HTML-page.

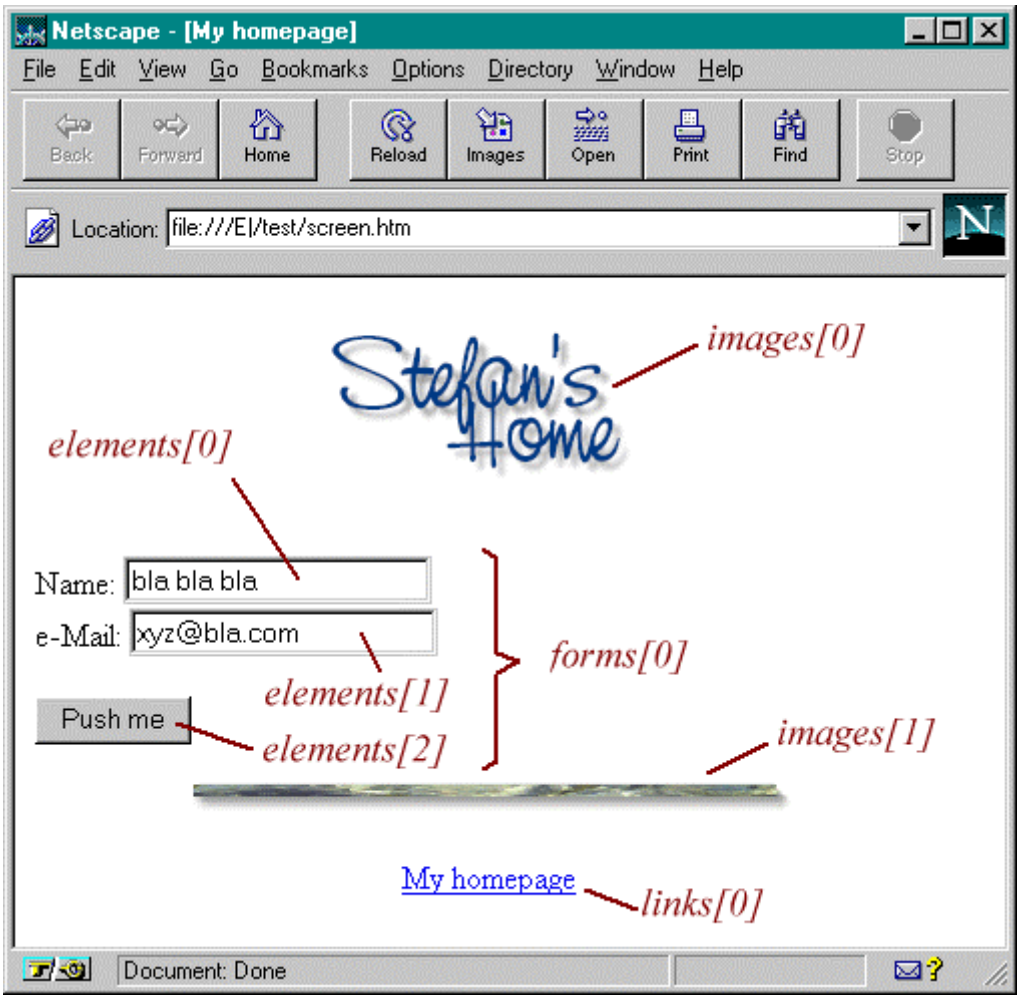
```
<html>
<head> </head>
<body bgcolor=#ffffff>

<center>
<img src="home.gif" name="pic1" width=200 height=100>
</center>
<p>
<form name="myForm">
Name: <input type="text" name="name" value="">
```

```
<br>
e-Mail:
<input type="text" name="email" value=""><br>
<br>
<input  type="button"  value="Push  me"  name="myButton"
onClick="alert('Yo')">
</form></p>

<p> <center>
<img src="ruler.gif" name="pic4" width=300 height=15>
</p>
<p><a  href="http://rummelplatz.uni-mannheim.de/~skoch/
"> My homepage</a> </center></p>
</body>
</html>
```

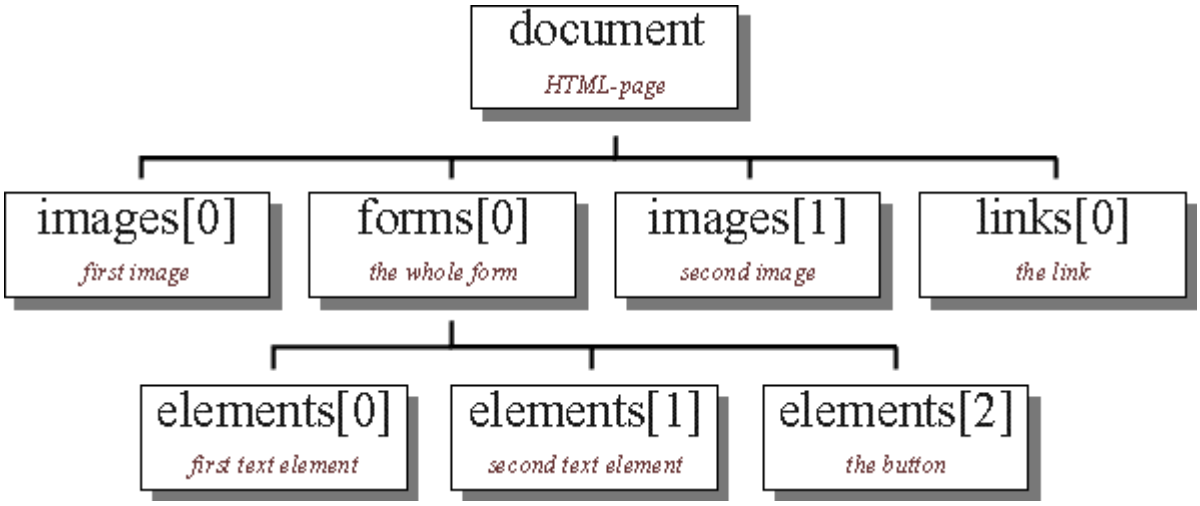
Here is a screenshot of this page:



We have two images, one link and a form with two text fields and a button. From JavaScript's point of view the browser window is a window-object. This window-object contains certain elements like the statusbar.

Inside a window we can load a HTML-document (or a file from another type - we will restrict ourselves to HTML-files). This page is a document-object, and it means the document-object represents the HTML-document which is loaded at the moment (the document-object is a very important object in JavaScript - it is used over and over again).

Properties of the document-object are for example the background color of the page. But what is more important is that all HTML-objects are properties of the document-object. A HTML-object is for example a link, or a form. The following image illustrates the hierachy created by the example HTML-page:



We want to be able to get information about the different objects and manipulate them. For this we must know how to access the different objects.

You can see the name of the objects in the hierarchy. If you now want to know how to address the first image on the HTML-page you have to look at the hierarchy. You have to start from the top. The first object is called document.

The first image the page is represented through **`images[0]`**. This means that we can access this object through JavaScript with **`document.images[0].`** If you want to know what the user entered into the first form element you must first think about how to access this object. Again we start from the top of our hierarchy. Follow the path to the object called **`elements[0]`** - put all the names of the object you pass together. This means you can access the first textelement through:

document.forms[0].elements[0]

But how can we now get to know the entered text? In order to find out which properties and methods an object offers you have to look into a JavaScript reference. There you will see that a textelement has got the property value. This is the text entered into the textelement. Now we can read out the value with this line of code:

```
name= document.forms[0].elements[0].value;
```

The string is stored in the variable name. We can now work with this variable. For example we can create a popup window with **alert("Hi " + name)**. If the input is **'Stefan**' the command **alert("Hi " + name)** will open a popup window with the text **'Hi Stefan'**.

If you have large pages it might get quite confusing by addressing the different objects with numbers - for example **document.forms[3].elements[17]** or was it **document.forms[2].elements[18]?**

To avoid this problem you can give unique names to the different objects. You can see in our HTML-code that we wrote for example:

```
<form name="myForm">
Name:
<input type="text" name="name" value=""><br>
...
```

This means that **forms[0]** is also called **myForm.** Instead of **elements[0]** you can use name (as specified with the name-property in the **<input>** tag). So instead of writing

```
name= document.forms[0].elements[0].value;
```

we can write the following

```
name= document.myForm.name.value;
```

This makes it much easier - especially with large pages with many objects. (Please note that you have to keep the same case - this means you cannot write **myform** instead of **myForm**).

Many properties of JavaScript-objects are not restricted to read-operations. You can assign new values to these properties. For example you can write a new string to a textelement.

Here is the code for this example - the interesting part is inside the onClick-property of the second **<input>** tag:

```
<form name="myForm">
<input type="text" name="input" value="bla bla bla">
<input type="button" value="Write"
  onClick="document.myForm.input.value= 'Yo!'; ">
```

There are lots of details to be explained. It was written just a small example. There you will see the use of different objects. Try to understand the script with the help of Netscape's documentation:

Here is the source code:

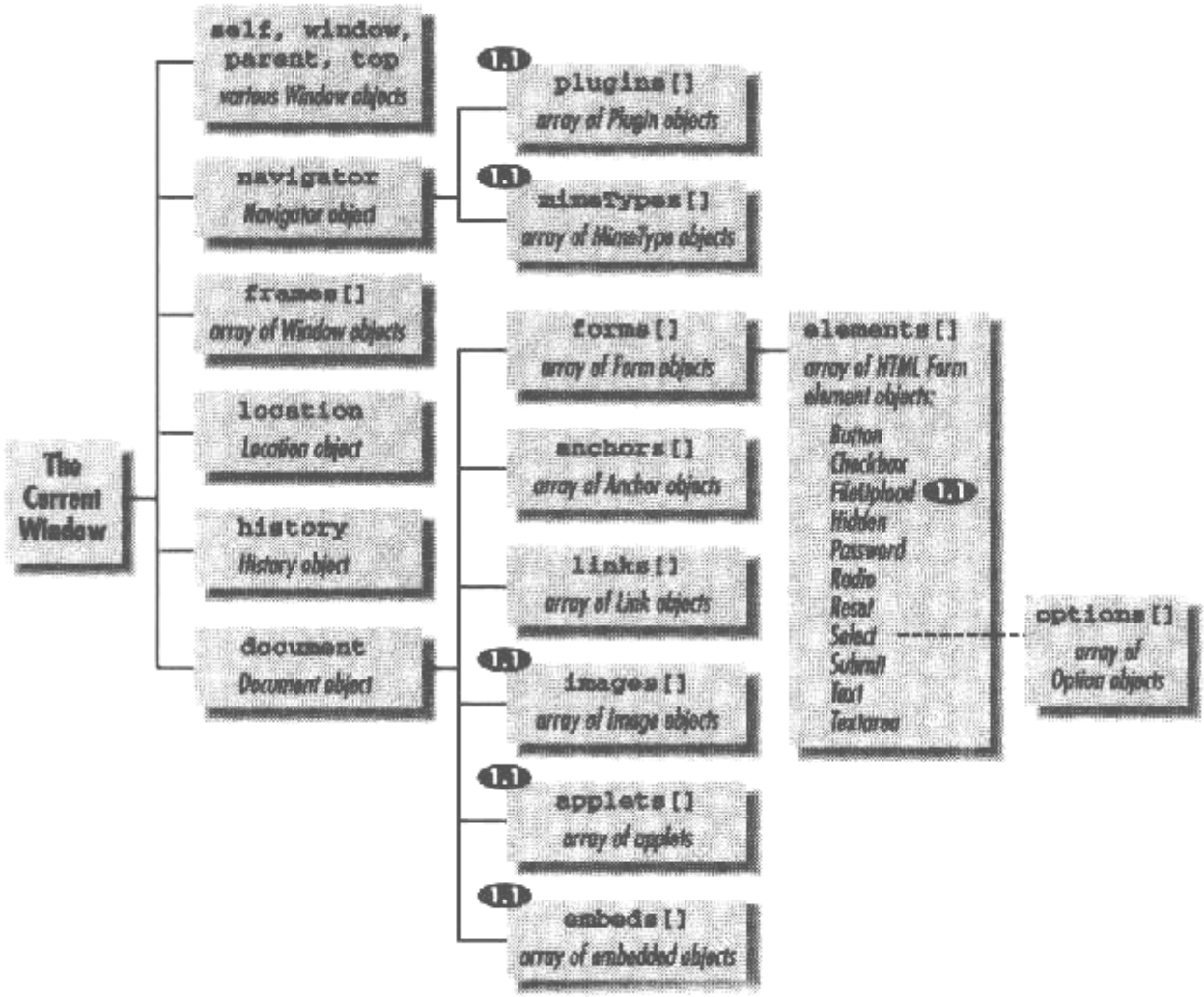
```
<html>
<head>
<title>Objects</title>
<script language="JavaScript">
<!-- hide
function first() {
  // creates a popup window with the
  // text which was entered into the text element
  alert("The value of the textelement is: " +
    document.myForm.myText.value);
}
function second() {
  // this function checks the state of the checkbox
  var myString= "The checkbox is ";
  // is checkbox checked or not?
  if  (document.myForm.myCheckbox.checked)  myString+=
"checked"
    else myString+= "not checked";
  // output string
  alert(myString);
}
// -->
</script>
</head>
<body bgcolor=lightblue>
```

```
<form name="myForm">
<input type="text" name="myText" value="bla bla bla">
<input type="button" name="button1" value="Button 1"
  onClick="first()">
<br>
<input type="checkbox" name="myCheckbox" CHECKED>
<input type="button" name="button2" value="Button 2"
  onClick="second()">
</form>
<p><br><br>
<script language="JavaScript">
<!-- hide
document.write("The background color is: ");
document.write(document.bgColor + "<br>");
document.write("The text on the second button is: ");
document.write(document.myForm.button2.value);
// -->
</script>
</body>
</html>
```

The location-object

Besides the window- and document-objects there is another important object: the location-object. This object represents the address of the loaded HTML-document. So if you loaded the page **`http://www.xyz.com/page.html`** then **`location.href`** is equal to this address. What is more important is that you can assign new values to **`location.href`**. This button for example loads a new page into the actual window:

```
<form>
<input type=button value="Yahoo"
  onClick="location.href='http://www.yahoo.com';">
</form>
```


The Current Window
self, window, parent, top
various Window objects
navigator
Navigator object
1.1
plugins[]
array of Plugin objects
1.1
mimeTypes[]
array of MimeType objects
frames[]
array of Window objects
location
Location object
history
History object
document
Document object
forms[]
array of Form objects
elements[]
array of HTML Form element objects:
Button
Checkbox
FileUpload 1.1
Hidden
Password
Radio
Reset
Select
Submit
Text
Textarea
options[]
array of Option objects
anchors[]
array of Anchor objects
links[]
array of Link objects
1.1
images[]
array of Image objects
1.1
applets[]
array of applets
1.1
embeds[]
array of embedded objects