Einführung in JavaScript

M. Duffner

Inhaltsverzeichnis

1. Literaturliste „Interaktive Web-Seiten“

ABTS, DIETMAR: *Grundkurs Java. Eine Einführung in das objektorientierte Programmieren mit Beispielen und Übungsaufgaben*. Braunschweig/Wiesbaden: Vieweg, 1999. [Math K 750:226]

FLANAGAN, DAVID: *Java in a nutshell*. o'Reilly, 3. Auflage 2000.

FLANAGAN, DAVID: *Javascript. Das umfassende Referenzwerk*. o'Reilly, 2. Auflage 2002. [EDV H 200: 1]

GOLL J.; WEIß C.; ROTHLÄNDER, P.: *Java als erste Programmiersprache. Java 2 Plattform*. Stuttgart: Teubner, 2000 2. Aufl. [Math K 750: 241 b]

HAELIER R.G.; FAHNENSTICH K.: Web Pages. HTML. Tools & Programme. Flash 4. JavaScript. Poing: Franzis, 2001. [Math K 750: 299]

HIRSEMANN, TH.; ROCHUSCH D.: *JavaScript. Wissen, das sich auszahlt.* Berlin: SPC TEIA Lehrbuch Verlag, 2003. [EDV H 200:6a]

HUNT, CRAIG: *TCP/IP Netzwerk-Administration*. o'Reilly, 2. Auflage 1998.

KOCH, DANIEL: *JavaScript lernen. Anfangen, anwenden, verstehen.* München: Addison-Wesley, 2003. [EDV H 200:2]

KOCH, STEFAN : *JavaScript. Einführung, Programmierung und Referenz*. Heidelberg: dpunkt.verlag, 2007 4. Aufl. [EDV H 200:19]

LEMAY, LAURA; CADENHEAD, R.: *Java 2 in 21 Tagen. Programmieren lernen mit Suns J2SE Version 1.3.2.* Markt + Technik Verlag, 2001.

LINDHORST,ARNO: *Das Einsteigerseminar CGI-Skriptprogrammierung*. Kaarst: bhv, 1999.

LOUIS, DIRK; MÜLLER, PETER: *Jetzt lerne ich Borland JBuilder 4. Der einfache Einstieg in die Java-Programmierung.* Markt + Technik Verlag, 2001.

MINTERT, STEFAN : *JavaScript 1.2 - Einführung, Referenz, Praxislösungen*. München: Addison-Wesley, 1998.

MINTERT, STEFAN; LEISEGANG, CHRISTOPH : *Ajax. Grundlagen, Frameworks und Praxislösungen*. Heidelberg: dpunkt.verlag, 2007 [EDV H 200:18].

MÜNZ, STEFAN: *Professionelle Websites. Programmierung, Design und Administration von Webseiten.* München: Addison-Wesley, 2005. [EDV P 10:10]

RRZN/UNIVERSITÄT HANNOVER (Hrsg.): *JavaScript 1.5 - Grundlagen*, 2003.

RINGMAYR TH. G.: *MS Frontpage. Wissen, das sich auszahlt.* Berlin: SPC TEIA Lehrbuch Verlag, 2002.

SIEGEL, DAVID: *Web Site Design. Killer Web Sites*. München: Markt & Technik, 1999.

SCHNEIDER, GERHARD: Eine Einführung in das Internet. Informatik Spektrum 18 (1995), 263-271.

SEEBOERGER-WEICHSELBAUM, MICHAEL: *Das Einsteigerseminar JavaScript*. Kaarst: bhv, 1998.

SOLYMOSI, A.; SCHMIEDECKE, I.*: Programmieren mit Java. Das Lehrbuch zum sicheren Umgang mit Objekten.* Braunschweig/Wiesbaden: Vieweg, 1999. [Math K 750: 199]

SPAINHOUR, STEPHEN; ECKSTEIN ROBERT: *Webmaster in a nutshell.* o'Reilly, 1999 second edition.

STEYER, RALPH.: *JavaScript. Webprogrammierung mit JavaScript, (X)HTML, CSS und Co.* Markt + Technik Verlag, 2005. [EDV H 200:11]

TURAU, VOLKER: *Techniken zur Realisierung Web-basierter Anwendungen*. Informatik-Spektrum 22 (1999), 3-12.

2. Linkliste zu HTML, CSS, XML, JavaScript, Java, PHP, Web-Design, ...

http://home.ph-freiburg.de/duffner/htm_javascript_java.htm

3. Was ist JavaScript?

JavaScript ist eine von Netscape entwickelte einfache Programmiersprache, die 1995 in den Browser von Netscape eingebunden wurde und rasch eine große Verbreitung fand. JavaScript wurde für den Einsatz im World-Wide-Web konzipiert und erweitert die Möglichkeiten der statischen Seitenbeschreibungssprache HTML. Mit JavaScript kann man interaktive und dynamische Web-Seiten erzeugen, die z.B. Anwendereingaben in einem Formular während der Eingabe überprüfen. JavaScript kann auch Ereignisse entgegennehmen und beispielsweise auf Mausklick eine Grafik austauschen.

JavaScript wurde nach 1995 ständig erweitert und Microsoft implementierte im Internet Explorer eine eigene JavaScript-Version, die aus lizenzrechtlichen Gründen JScript genannt wurde. Um dem Auseinanderdriften der beiden Versionen von Netscape und Microsoft entgegenzuwirken, wurde 1997 der Standard ECMAScript von der europäischen Standardisierungsorganisation ECMA (European Computer Manufacturers Association) verabschiedet. Dies führte in der Folge wieder zu einer Angleichung der beiden Sprachen, so dass sie sich heute nur noch in Details unterscheiden.

JavaScript hat bei der so genannten Ajax-Technologie im Web eine zentrale Bedeutung. Ajax ist ein Akronym für **A**synchronous **J**avaScript **a**nd **X**ML. Mit Ajax werden im Web interaktive, Desktop-ähnliche Anwendungen realisiert, bei denen Serveranfragen aus Anwendersicht in den Hintergrund treten. Bei der asynchronen Datenübertragung zwischen einem Webserver und dem Browser wird in einer HTML-Seite eine HTTP-Anfrage durchgeführt, ohne dass die Seite komplett neu geladen werden muss. JavaScript dient dabei zur Manipulation des Document Object Models (DOM; vgl. S. 22) und zur dynamischen Darstellung der Inhalte. Diese Techniken werden auch als Grundlage für eine neue Generation des Webs gesehen, die derzeit auch mit dem Begriff „Web 2.0“ gehandelt werden.

Die Namensähnlichkeit von Java und JavaScript legt eine enge Verwandtschaft der beiden Programmiersprachen nahe. JavaScript lehnt sich zwar in der Syntax an die von Sun Microsystems entwickelte Programmiersprache Java an, steht aber in der Funktionalität hinter Java zurück. Java ist im Unterschied zu JavaScript eine vollwertige Programmiersprache, die grundsätzlich für alle Anwendungsfelder eingesetzt werden kann. Java ist objektorientiert. JavaScript wird als objektbasiert bezeichnet. JavaScript verfügt zwar über ein Objektkonzept, arbeitet aber im Wesentlichen nur mit HTML-Dokumenten und deren Elemente.

Java verlangt vom Programmierer eine strenge Typisierung der Variablen. Bei der Definition von Variablen muss genau angegeben werden, um welchen Typ (Ganzzahl, Fließkommazahl, Zeichen, Zeichenkette, ...) es sich handelt. Bei JavaScript ist dies nicht notwendig. Das mag auf den ersten Blick leichter erscheinen, kann aber auch fehleranfälliger sein.

JavaScript-Programme werden wahlweise direkt in der HTML-Datei oder in separaten Dateien notiert. Sie werden nicht - wie etwa Java-Programme - compiliert, sondern als Quelltext zur Laufzeit interpretiert. Dazu benötigen Browser entsprechende Interpreter-Software. In der Regel ist diese in aktuellen Browsern integriert. Der Browser muss aber so konfiguriert sein, dass die Ausführung von JavaScripts zugelassen ist.

4. Einbinden von JavaScript in HTML

```
<script type="text/javascript">
  < !--
      JavaScript-Anweisungen
  //-->
</script>
<noscript>Dein Browser kann leider kein JavaScript!</noscript>
```

Mit <script type="text/javascript"> leiten Sie einen Bereich für JavaScript innerhalb einer HTML-Datei ein. Dahinter - am besten in der nächsten Zeile - können Sie mit **<!--** einen HTML-Kommentar einleiten. Dadurch erreichen Sie, dass nicht JavaScript-fähige Browser, die JavaScript nicht kennen, den folgenden

JavaScript-Code ignorieren und nicht irrtümlich als Text innerhalb der HTML-Datei interpretieren. Die <script>-Tags werden dann i.d.R. überlesen. Da heute die meisten Browser JavaScript-fähig sind, kann diese Sicherheitsmaßnahme auch weggelassen werden.

Am Ende eines JavaScript-Bereichs schließen Sie ggf. mit --> den HTML-Kommentar. Damit JavaScript wiederum nicht über das --> stolpert, setzt man zuvor ein JavaScript-Kommentar (vgl. 7.) und schreibt insgesamt //-->. Mit </script> wird der Bereich für den JavaScript-Programmcode geschlossen.

Früher wurde für <script type="text/javascript"> oft <script language="JavaScript"> verwendet. Der HTML-Standard schreibt aber die erstgenannte Version vor, die auch von allen aktuellen Browsern unterstützt wird.

Skripte können auch *extern eingebunden* werden:

```
<script type=•text/javascript•
       src= •…/JavaScript_Skripte/meinSkript1.js•>
</script>
```

Im Beispiel wird die Datei „meinSkript1.js" aus dem Unterverzeichnis "…/JavaScript_Skripte geladen. Das Attribut "text/javascript" teilt dem Interpreter mit, dass die Datei das Skript als einfachen Text enthält.

5. Stellung von JavaScript-Bereichen in einer HTML-Datei

Es gibt keine festen Vorschriften dafür, an welcher Stelle einer HTML-Datei ein JavaScript-Bereich definiert werden muss. JavaScript-Code kann sowohl im Header einer HTML-Seite stehen, wie im Body der Seite:

```
<html>
  <head>
       HTML-Kopfinformationen, wie z.B. </title>
              JavaScript-Funktionsdefinitionen
  </head>
  <body>
       HTML
              JavaScript-Anweisungen
       HTML
              JavaScript-Anweisungen
       HTML
  </body>
</html>
```

6. Das 1. Programm: „Hallo Welt!"

Im folgenden Beispiel wird mit Hilfe von JavaScript ein Meldungsfenster mit dem Text "Hallo Welt!" auf dem Bildschirm ausgegeben.

```
<html>
  <head>
    <title>Test</title>
  </head>
  <body>
    <script type=•text/javascript•>
      <!--
      alert("Hallo Welt!");
      //-->
    </script>
    <noscript>Dein Browser kann leider kein JavaScript!</noscript>
  </body>
</html>
```

7. Kommentare

Kommentare in HTML und Kommentare in JavaScript werden ganz verschieden notiert und haben nichts miteinander zu tun.

JavaScript
Einzeiliger Kommentar, meist am Zeilenende einer JavaScript-Anweisung: durch // eingeleitet.

```
alert(„Hallo Welt“);    // Das ist eine Ausgabeanweisung
Mehrzeiliger Kommentar: durch    /*    */       eingeschlossen.
/*
Hier kann der Kommentar stehen,
der über mehrere Zeilen geht
*/
```

HTML

```
<!--
Hier steht der Kommentar. Oft ist dies auch JavaScript-Code.
-->
```

Die Anwendung des HTML-Kommentars im Inneren des <script>-Containers kann in einigen Browsern zu Problemen führen, die auf unglückliche Programmierung dieser Browser-Versionen zurückzuführen sind. Die Zeichen `--` stehen in JavaScript für einen Operator und bei diesen Browsern wird nicht erkannt, dass --> das Ende eines HTML-Kommentars darstellt. Der JavaScript-Interpreter der Browsers versucht, die JavaScript-Anweisung -- auszuführen. Die Folge ist ein JavaScript-Fehler und der HTML-Kommentar wird evtl. nicht geschlossen. Man versteckt daher das Ende des HTML-Kommentars hinter einem JavaScript-Kommentar:

`//-->` (vgl. „Einbinden von JavaScript in HTML, Seite 5)

8. JavaScript-Elemente in HTML-Tags

JavaScript kann auch innerhalb herkömmlicher HTML-Tags vorkommen. Das ist dann i.d.R. kein komplexer Programmcode, sondern in der Regel nur der Aufruf bestimmter Methoden, Funktionen, Objekte, Eigenschaften. Für den Aufruf gibt es sogenannte **Event-Handler**. Das sind JavaScript-eigene Attribute in HTML-Tags. Für jeden der möglichen Event-Handler ist festgelegt, in welchen HTML-Tags er vorkommen darf.

Beispiel:

```
<html>
  <head>
    <title>Hallo Welt mit Button-Ereignis</title>
  </head>
   <body>
   <form>
     <input type="button" value="Dr&uuml;ck mich!"
             onClick="alert('Autsch!')">
    </form>
  </body>
</html>
```

In der HTML-Datei ist ein Formular mit einem Button definiert. Wenn der Anwender auf den Button klickt, wird ein so genanntes „Ereignis“ ausgelöst. Ereignisse werden von *Event-Handlern* entgegengenommen. Der Event-Handler „onClick“ sorgt dafür, dass der JavaScript-Code "alert('Autsch!')" abgearbeitet wird.

Im folgenden Beispiel wird beim Verlassen des Bereiches mit der Maus, den der Link mit dem URL http://www.ph-freiburg.de auf einer Seite einnimmt, die Meldung „Wollen Sie nicht auf die PH-Homepage?“ mit alert() ausgegeben:

```
<html>
  <head>
    <title>Hallo Welt 7</title>
    <meta http-equiv=•Content-Script-Type• content=•text/javascript• />
  </head>
  <body>
    <a href="http://www.ph-freiburg.de"
    onMouseOut="alert('Wollen Sie nicht auf die PH-Homepage?')">
    PH-Homepage</a>
   </body>
</html>
```

Im obigen Beispiel wurde mit einem <meta>-Tag und den Eigenschaften http-equiv und content als Standardskriptsprache der Web-Seite JavaScript festgelegt. Damit ist eindeutig, dass nach Eintreten des onMouseOut-Ereignis JavaScript-Code folgen soll. Oft wird diese Festlegung weggelassen und darauf vertraut, dass der Browser JavaScript als Standard verwendet.

Im folgenden Beispiel wird beim Ereignis Klick auf einen Button eine mit JavaScript im Header definierte Funktion aufgerufen:

```
<html>
  <head>
  <script language="JavaScript">
  <!--
   function Quadrat(Zahl)
    {
     Ergebnis = Zahl * Zahl;
     alert("Das Quadrat von " + Zahl + " ist " + Ergebnis + ".");
    }
  //-->
  </script>
  </head>

  <body>
  <form>
    Geben Sie eine Zahl ein:
    <p><input type="text" name="Eingabe" size="20"></p>
    <p><input type="button" value="Quadrat berechnen"
    onClick="Quadrat(Eingabe.value)"></p>
  </form>
  </body>
  </html>
```

Im Kopf der HTML-Datei wird hier eine Funktion Quadrat definiert, die als Übergabe eine Zahl als Parameter erwartet. Von der übergebenen Zahl berechnet die Funktion das Quadrat. Die Funktion wird durch den Event-Handler onClick aufgerufen und als Parameter wird der Wert des Texteingabefeldes „Eingabe“ des Formulars übergeben (zu Funktionen vgl. Seite 17ff).

Im Folgenden finden Sie eine Auflistung der in JavaScript zur Verfügung stehenden Event-Handler.

9. Event-Handler

Event-Handler	Ereignis	HTML-Tags
onAbort	beim Abbrechens des Ladens einer Grafikdatei	<img>
onBlur	beim Verlassen eines Texteingabefeldes	<body> <frameset> <input> <layer> <select> <textarea>
onChange	bei geändertem Inhalt von Formularelementen	<input> <select> <textarea>
onClick	beim Anklicken von Elementen	<a> <area> <input> <textarea>
onDblClick	beim Doppelklicken auf Elemente	<a> <area> <input> <textarea>

onError	bei fehlerhaften Grafiken	<img>
onFocus	bei Aktivierung eines Elementes durch den Anwender	<body> <frame> <input> <layer> <select> <textarea>
onKeyDown	bei gedrückter Taste, wenn der Anwender ein Element aktiviert hat	<input> <textarea>
onKeyPress	Anwender drückt eine Taste und hält diese gedrückt.	<input> <textarea>
onKeyUp	Anwender lässt eine gedrückte Taste wieder los.	<input> <textarea>
onLoad	beim Laden eines Dokumentes	<frameset> <body>
onMouseDown	bei gedrückter Maustaste	<input type="button"> <a>
onMousemove	bei weiterbewegter Maus	<input type="button"> <a>
onMouseOut	beim Verlassen eines definierten Bereichs mit der Maus	<a> <area>
onMouseOver	beim Eintreten der Maus in einen definierten Bereich	<a> <area>
onMouseUp	bei losgelassener Maustaste	<input type="button"> <a>
onReset	beim Zurücksetzen von Formularelementen	<form>
onSelect	bei getroffener Auswahl in Select-Menüs	<input> <textarea>
onSubmit	beim Absenden von Formulardaten	<form>
onUnload	beim Verlassen einer Web-seite	<frameset> <body>

10. Grundelemente der Sprache JavaScript

10.1. Datentypen

JavaScript kennt folgende einfache Datentypen[1]:

1. Zahlen (Numbers)
2. Boolesche Werte (Boolean)
3. Zeichenketten (Strings)

Diese Datentypen stehen einmal als einfache Werte zur Verfügung, aber auch als Objekte (vgl. „Einige wichtige vordefinierte Objekte mit ihren Eigenschaften, Methoden und Unterobjekten“).

10.1.1. Zahlen (Numbers)

JavaScript kennt ganze Zahlen (negative und positive ganze Zahlen wie ... -3, -2, -1, 0, 1, 2, 3, ...) und Fließkommazahlen (Dezimalzahlen). Ganze Zahlen haben 32 Bit Speicherplatz zur Verfügung. Ein Bit wird für das Vorzeichen verwendet. Damit gibt es zwei mal 2^{31} = 2.147.483.648 ganze Zahlen: - 2147483647, …-1, 0, 1, …, 2147483648. Für Fließkommazahlen werden 8 Byte verwendet und deren Gültigkeitsbereich erstreckt sich von +/-2,2250738585072014 · 10^{-308} bis 1,7976931348623157 · 10^{308}.

10.1.2. Boolesche Werte (Boolean)

Dieser Datentyp besteht aus den Wahrheitswerten *true* (wahr) und *false* (falsch). Boolesche Werte werden auch bei Ausdrücken mit Vergleichsoperatoren zurückgeliefert. Z.B. besitzt der Ausdruck "5 < 2" den Wahrheitswert *false*. Wichtig sind solche Ausdrücke bei Fallunterscheidungen, bei denen i.d.R. ein Ausdruck mit einer Variablen als Bedingung verwendet wird – z.B. „if (Eingabe >=12) …“.

[1] Darüber hinaus kennt JavaScript noch die Datentypen *function* und *object*, auf die später eingegangen wird.

10.1.3. Zeichenketten (Strings)

Eine Zeichenkette ist eine Folge von Zeichen. Zeichenketten werden in JavaScript in doppelte oder einfache Anführungszeichen eingeschlossen: "Hallo" oder 'Hallo Welt!'. Innerhalb eines Strings sind alle am Bildschirm darstellbaren Zeichen erlaubt. Es gibt einige Sonderzeichen, die in Strings mit einem Backslash kodiert werden müssen:

\b	Backspace	\f	Form Feed (Seitenvorschub)	\n	New Line (Neue Zeile)
\r	Carriage Return (Wagenrücklauf)	\t	Tab (Tabulator)	\"	Anführungszeichen
\'	Hochkomma	\\	Backslash		

Beispiele für korrekte Zeichenketten:

```
"blablabla"
'Er sagte, "Guten Tag!"'
"Er sagte, \"Guten Tag!\""
""  (leerer String)
```

Es gibt theoretisch keine Längenbegrenzung für Strings wie in anderen Programmiersprachen. Natürlich sind aber durch das Betriebssystem Grenzen gesetzt. In der Praxis sollten aber Strings bis 64 KiloByte möglich sein.

10.2. Variablen und Namen

Die im vorangegangen Abschnitt vorgestellten Ausdrücke, wie Strings und Zahlen, sind Konstanten *(Literale*) mit einer unveränderlichen Bedeutung. *Variablen* sind Speicherbereiche, in denen Daten, die während der Ausführung eines Programms benötigt werden, gespeichert werden können. Die in einer Variablen gespeicherten Daten werden als der *Wert* der Variablen bezeichnet. Der Wert einer Variablen kann im Unterschied zu Konstanten während der Laufzeit eines Programms jederzeit geändert werden. Mit welchen Werten eine Variable belegt werden kann und welche Operationen mit den Variablen zulässig sind, wird durch den Typ der Variablen bestimmt. Um mit Variablen arbeiten zu können, sollten die benötigten Variablen zuerst definiert werden.

Ein Beispiel:

```
var text = "Hallo";
alert(text);
```

In der ersten Zeile dieses Teils eines JavaScript-Programms wird eine Variable definiert, die den Namen text hat und dieser Variablen text wird der Wert "Hallo" zugewiesen. Die Wertzuweisung wird mit dem *Zuweisungsoperator* **=** durchgeführt. "Hallo" ist ein String, und damit ist auch der Typ der Variablen festgelegt. Im Unterschied zu anderen Programmiersprachen ist in JavaScript keine explizite Typisierung möglich – z.B. nach dem Muster number var x, um den Typ Number zu vereinbaren. Die Typisierung geschieht erst durch die Wertzuweisung var x = 5. Die Wertzuweisung var x = “5“ würde dagegen eine String-Variable typisieren!

JavaScript unterscheidet Groß- und Kleinschreibung, d.h. die Variable mit Namen „text“ ist eine andere Variable als die mit dem Namen „Text“.

Im folgenden Beispiel werden die Festlegung unterschiedlicher Datentypen der Variablen und die nachfolgende Anweisung zu einem Problem führen:

```
var x = "Hallo";
alert(x);
var y = 5;
alert(x / y);
```

Die Variable x ist vom Datentyp *String* und die Variable y vom Typ *Number*. Der Ausdruck "x/y" (x dividiert durch y) sollte also zu einer Fehlermeldung führen. JavaScript führt zwar noch die Anweisung "alert(...)" aus. Aber in dem von der Funktion alert erzeugten Meldungsfenster erscheint "*NaN*" - was für

"*Not a Number*" steht. Andere Programmiersprachen würden schon vor der Ausführung des Programms mit einem Hinweis auf die Unverträglichkeit der beiden Datentypen abbrechen. In JavaScript kann auch jederzeit eine Variable, die z.B. mit einem String-Wert initialisiert wurde, wieder mit einer Zahl als Wert belegt werden, was bei anderen Programmiersprachen nicht möglich ist. So führt das folgende Programm zu keiner Fehlermeldung:

```
var celsius = "text";
celsius = 100
var fahrenheit = 9/5 * celsius + 32
alert(fahrenheit);
```

JavaScript ist also sehr "tolerant". Auch *var* vor einer Wertzuweisung kann weggelassen werden. Nur die Benutzung einer Variablen, der noch kein Wert zugewiesen wurde, erzeugt einen Fehler. Diese „Toleranz“ von JavaScript wird oft mit dem Begriff „loose typing“ typisiert, im Unterschied zum „strong-typing“ anderer Programmiersprachen.

"Guter" Programmierstil ist es, die Variablen mit der Anweisung *var* zu definieren, mit einem Wert zu initialisieren und mit einem Kommentar über die Bedeutung zu versehen:

```
var summand1;       // erster Summand, Number-Typ
var summand2;       // zweiter Summand, Number-Typ
var summe = 0;      // Ergebnis Addition, Number-Typ mit 0 initialisiert
summand1 = 3;
summand2 = 4;
summe = summand1 + summand2
```

Damit wird die Lesbarkeit des Programms verbessert. Die Initialisierung von Variablen ist wichtig, um ungewollte Nebeneffekte zu vermeiden. Die Deklaration der Variablen und die Wertzuweisung hätte man auch für summand1 und summand2 in eine Zeile packen können, z.B. var summand1 = 3.

Für die Vergabe von Namen für Variablen oder Funktionen, auch *Bezeichner* genannt, gelten bestimmte Regeln:

- Namen beginnen mit einem Buchstaben oder einem Unterstrich (_). Deutsche Sonderzeichen, wie Umlaute sollten trotz der zu erwartenden "Toleranz" von JavaScript nicht verwendet werden.
- Als weitere Zeichen sind auch Ziffern (0, 1, …, 9) zugelassen.
- Reservierte Wörter, wie z.B. else, function, while, … dürfen natürlich nicht benutzt werden.

Der *Gültigkeitsbereich* einer Variablen, die im Hauptprogramm definiert wurde, umfasst das gesamte Programm. Man spricht von *globalen Variablen*, die an jeder Stelle des Programms verwendet werden können. Im Hauptprogramm gibt es Funktionen, die einmal definiert werden und später im Hauptprogramm oder von anderen Funktionen ausgeführt werden können (vgl. „Funktionen“). Innerhalb einer Funktion ist es möglich *lokale Variablen* zu verwenden, die im Hauptprogramm nicht bekannt sind und nur innerhalb der Funktion verwendet werden können.

10.3. Ausdrücke und Operatoren

Ausdrücke sind in der einfachsten Form eine einzelne Konstante oder eine einzige Variable oder eine Kombination aus Variablen und Konstanten und Operatoren, die sie verknüpfen. Jeder Ausdruck besitzt einen Wert aus dem Bereich der Datentypen *Number*, *String* oder *Boolean*:

Ausdruck	**Beispiele**	**Typ des Ausdrucks**	**Wert des Ausdrucks**
ohne Operator	7	Number	7
ohne Operator	true	Boolean	true
ohne Operator	"Hallo Welt"	String	"Hallo Welt"
mit Operator	a = 7	Number	7
mit Operator	8 < 7	Boolean	false

mit Operator	a < 8	Boolean	true (für a = 7) false (für a = 1001)
mit Operator	x = (3 + 2) * 4	Number	20
mit Operator	y = "Hallo" + "Welt"	String	"HalloWelt"

In der Tabelle werden Operatoren benutzt, die (in der Regel zwei) Variablen oder Konstanten kombinieren und ein Ergebniswert ermitteln. Operatoren lassen sich analog zu den Datentypen unterteilen in arithmetische, Zeichenketten- und boolesche Operatoren. Operatoren müssen zu den Operanden passen. Die Division von Strings macht beispielsweise keinen Sinn.

10.4. Operatoren in JavaScript:

Operator-Kategorie	Operator	Beschreibung
Arithmetische Operatoren	+	(Addition) Addiert 2 Zahlen. Beispiel: x+2.1
	++	(Inkrement) Addiert eins zu einer Variablen. Beispiel: x++
	-	(Subtraktion) Subtrahiert zwei Zahlen. Beispiel: x-2.1
	--	(Dekrement) Subtrahiert eins von einer Variablen. Beispiel: x--
	*	(Multiplikation) Multipliziert 2 Zahlen.
	/	(Division) Dividiert 2 Zahlen.
	%	(Modulo) Berechnet den Rest der ganzzahligen Division zweier Zahlen.
String-Operatoren	+	(String-Addition) Fügt zwei Strings zusammen.
	+=	Fügt zwei Strings zusammen und weist das Ergebnis dem ersten Operanden zu.
Logische Operatoren	&&	(Logisches UND) Gibt true zurück, falls beide logische Operanden true sind. Ansonsten wird false zurückgegeben.
	||	(Logisches ODER) Gibt true zurück, falls einer der beiden logischen Operanden true ist. Falls beide false sind wird false zurückgegeben.
	!	(Logische Negation) Falls der Operand true ist, wird false zurückgegeben. Ansonsten wird true zurückgegeben.
Zuweisungs-Operatoren	=	Wertzuweisung: Weist dem ersten Operanden den Wert des zweiten Operanden zu.
	+=	Addiert 2 Zahlen und weist der ersten das Ergebnis zu.
	-=	Subtrahiert 2 Zahlen und weist der ersten das Ergebnis zu.
	*=	Multipliziert 2 Zahlen und weist der ersten das Ergebnis zu.
	/=	Dividiert 2 Zahlen und weist der ersten das Ergebnis zu.
	%=	Berechnet modulo von 2 Zahlen und weist der ersten das Ergebnis zu.
	&=	Führt ein bitweises AND durch und weist das Ergebnis dem ersten Operanden zu.
	^=	Führt ein bitweises XOR durch und weist das Ergebnis dem ersten Operanden zu.
	|=	Führt ein bitweises OR durch und weist das Ergebnis dem ersten Operanden zu.
	<<=	Führt eine Links-Verschiebung durch und weist das Ergebnis dem ersten Operanden zu.
	>>=	Führt eine Vorzeichen-weiterreichende Verschiebung durch und weist das Ergebnis dem ersten Operanden zu.
	>>>=	Führt ein Null-füllende-Rechtsverschiebung durch und weist das Ergebnis dem ersten Operanden zu.

Vergleichs-operatoren	==	Gibt true zurück, falls die Operanden gleich sind.
	!=	Gibt true zurück, falls die Operanden nicht gleich sind.
	>	Gibt true zurück, falls der linke Operand größer ist als der rechte Operand.
	>=	Gibt true zurück, falls der linke Operand größer oder gleich ist wie der rechte Operand.
	<	Gibt true zurück, falls der linke Operand kleiner ist als der rechte Operand.
	<=	Gibt true zurück, falls der linke Operand kleiner oder gleich ist wie der rechte Operand.
Spezielle Operatoren	?:	Für einfaches "if...then...else"
	,	Evaluiert zwei Ausdrücke und gibt das Ergebnis des zweiten Ausdruckes zurück.
	delete	Zum Löschen einer Objekteigenschaft oder eines Elementes eines Arrays.
	new	Erstellt eine Instanz eines benutzerdefinierten Objekttyps oder eines in JavaScript bereits definierten Objekttyps.
	this	Schlüsselwort, um sich auf das aktuelle Objekt zu beziehen.
	typeof	Gibt den Typ des übergebenen Operanden zurück.
	void	Mit diesem Operator wird ein Ausdruck ausgewertet, ohne dass ein Wert zurückgeliefert wird.

Bitte unterscheiden Sie zwischen dem *Zuweisungsoperator* „ **=“** und dem *Vergleichsoperator* **„= =“**. Mit = wird einer Variablen ein Wert zugewiesen, z.B. a = 5. Der Vergleichsoperator vergleicht zwei Variablen (a = = x), eine Variable und ein Literal (a = = 5) oder zwei Literale (5 = = 5) und jeder dieser Ausdrücke hat als Wert entweder den Booleschen Wert false oder true.

10.5. Schleifen

Schleifen finden sich in jeder Programmiersprache. Sie erlauben es, eine Anweisungsfolge mehrfach zu wiederholen. JavaScript stellt 4 Typen von Schleifen zur Verfügung:

- *for*-Schleifen
- *while*-Schleifen
- *do-while*-Schleifen
- *for.. in*-Schleifen.

10.5.1. *for*-Schleife

For-Schleifen eignen sich, um eine Anweisungsfolge in einer *von vorneherein feststehenden Anzahl von Durchläufen* auszuführen.
Die allgemeine Syntax einer *for*-Schleife ist:

```
for(Startanweisung; Bedingung; Zählanweisung) {
  Anweisungsblock
}
```

Beispiel für ein Programm mit einer *for*-Schleife:

```
<html><head><title>Test</title>
</head><body>
<script type="text/javascript">
<!--
   for(i = 1; i <= 10; i++)
    {
     var x = i * i;
     document.write("<br>Das Quadrat von " + i + " ist " + x);
```

```
      }
// -->
</script>
</body></html>
```

Eine for-Schleife beginnt mit dem Wort *for*. Dahinter werden, in Klammern stehend, die Schleifenbedingungen formuliert. Bei der for-Schleife gilt dabei eine feste Syntax. Innerhalb der Schleifenbedingung werden drei Anweisungen notiert. In der ersten Anweisung wird ein Schleifenzähler definiert und initialisiert. Im Beispiel wird ein Zähler i definiert und mit dem Wert 1 initialisiert. Die zweite Anweisung enthält die Bedingung, ab der die Schleife beendet wird. Dazu benötigt man einen Vergleichsoperator. In der dritten Anweisung wird der Schleifenzähler so verändert, dass er irgendwann die in der zweiten Anweisung notierte Bedingung erfüllt. Im Beispiel wird i bei jedem Schleifendurchgang um 1 erhöht, so dass der Wert irgendwann gleich 10 ist und damit die Bedingung erfüllt ist. Anstatt „i++“ hätte man auch „i = i +1“ schreiben können.

Das Beispiel benutzt den Schleifenzähler, um bei jedem Schleifendurchgang das Quadrat des aktuellen Werts zu ermitteln. Das Ergebnis wird dann HTML-formatiert ins aktuelle Dokumentfenster des Browsers geschrieben. Dazu wird die Methode *write* des HTML-Objektes *document* benutzt.

while-Schleife

Eine while-Schleifen kommt in Betracht, wenn Sie nicht wissen, wie oft die Schleife durchlaufen werden soll. Mit diesem Schleifentyp werden die Anweisungen in der Schleife *solange wiederholt, bis die Bedingung, die am Anfang der Schleife (Eintrittsbedingung) formuliert wird, erfüllt ist.*

Die allgemeine Syntax einer *while*-Schleife lautet:

```
while(Bedingung) {
  Anweisungsblock
}
```

Eine while-Schleife beginnt mit dem Wort while (while = solange). Dahinter folgt, in Klammern stehend, die Eintrittsbedingung. Um eine Bedingung zu formulieren, brauchen Sie Vergleichsoperatoren. Der Inhalt der Schleife wird solange wiederholt, wie die Schleifenbedingung wahr ist. In der Regel enthält eine while-Schleife mehrere Anweisungen, die innerhalb der Schleife stehen. Alle Anweisungen der Schleife müssen innerhalb den geschweiften Klammern { und } stehen.

Beispiel für eine *while*-Schleife:

```
<html> <script type="text/javascript">
<!--
var fortsetzen = true;
var x = 1000;
while (fortsetzen) {
   x++;
   fortsetzen = confirm("x hat den Wert " + x + "\nSchleife
fortsetzen?");
  }
// -->
</script> </html>
```

In dem Beispiel wird die Schleife mindestens einmal durchlaufen und maximal solange durchlaufen bis der Benutzer im „confirm-fenster“ auf Abbrechen klickt. Als erstes wird der Wert „1001“ der Variablen x ausgegeben und danach, falls der Benutzer nicht abbricht, um eins nach oben gezählt und wieder ausgegeben. beachten Sie, dass die Variable auf die sich die Eintrittsbedingung der Schleife bezieht, in der Schleife selbst verändert wird. Im Beispiel ändert der Benutzer ggf. im Bestätigungsfenster den Wert der variablen *fortsetzen* auf false. Wenn die Variablen der Abbruchbedingung nicht innerhalb der Schleife geändert werden, kann die Schleife nicht beendet werden und Sie haben eine so genannte „Endlosschleife“.

10.5.2. *do*-while-Schleife

Die allgemeine Syntax einer *do-while*-Schleife ist:

```
do {
  Anweisungsblock
  }
while(Bedingung)
```

Eine while-Schleife prüft zuerst, ob die Schleifenbedingung (Eintrittsbedingung) erfüllt ist. Falls die Bedingung nicht erfüllt ist, wird die Schleife nicht ausgeführt. Daher kann es sein, dass die Schleife gar nicht ausgeführt wird. Bei einer *do-while-Schleife* wird dagegen der Anweisungsblock auf jeden Fall einmal ausgeführt. Erst am Ende der Schleife wird getestet, ob die Anweisungen wiederholt werden soll.

Beispiel für eine *do-while*-Schleife:

```
<html>
<script type="text/javascript">
<!--
var x = 5;
do {
      document.write("<br>x * x = " + (x * x));
      x = x + 1;
   }
while(x<5);
// -->
</script>
</html>
```

Im Beispiel wird die Schleife einmal durchlaufen, obwohl x schon den Wert 5 hat, und es wird das Quadrat von 5 ausgegeben.

10.5.3. *for..in*-Schleife

Die for..in-Schleife wird meist im Zusammenhang mit Objekten verwendet. Mit diesem Schleifentyp können z.B. alle Eigenschaften eines Objekts ermittelt werden. Objekte werden im Skript später behandelt. Daher unten nur ein Beispiel im Vorgriff auf den noch zu behandelnden Stoff.
Die allgemeine Syntax einer *for..in*-Schleife ist:

```
for Eigenschaft in Objekt) {
  Anweisungsblock
  }
```

Beispiel für eine *for..in*-Schleife:

```
<html>
<script type="text/javascript">
<!--
 var wert = "";
 for (var Eigenschaft in navigator)
   wert = wert + Eigenschaft + ": " + navigator[Eigenschaft] + "<BR>";
 document.write(wert);
// -->
</script>
</html>
```

Im Beispiel werden die Eigenschaften des Navigator-Objektes (Browser) ausgegeben.

10.5.4. *break*- und continue-Anweisung

Die Anweisungen *break* und *continue* bieten die Möglichkeit, eine Schleife vorzeitig abzubrechen. *Break* verlässt eine Schleife und setzt die Bearbeitung des Scripts mit der ersten Anweisung nach der Schleife fort. *Continue* beendet den aktuellen Schleifendurchlauf und setzt das Script mit dem nächsten Schleifendurchlauf fort. Solche „Sprung-Anweisungen“ (ähnlich wie goto in Basic) verhindern die Lesbarkeit von Programmen und sind häufig die Ursache für Fehler. Sie sollten m.E. daher nicht eingesetzt werden.

10.6. Fallunterscheidungen

JavaScript stellt die gängigen Typen für Fallunterscheidungen zur Verfügung:

- *if*-Anweisung (mit optionalem *else*-Zweig)
- *switch*- Anweisung

10.6.1. *if-Anweisung* (mit optionalem else-Zweig)

Für die einmalige bedingte Ausführung von Programmteilen gibt es die *if*- Anweisung. Die Syntax der *if* - Anweisung lautet:

```
if (Bedingung) {
  Anweisungsblock
                    }
else {
  Anweisungsblock
                    }
```

Die *if*-Anweisung kann nahezu wörtlich übersetzt werden: Wenn die Bedingung erfüllt ist, führe den Anweisungsblock bzw. die „*dann-anweisung(en)*“ aus, im anderen Fall den Anweisungsblock mit den „*sonst-anweisung(en)*“. Der *else*-Zweig ist optional.

Ein einfaches Beispiel für eine *if*-Anweisung ohne *else*-Zweig:

```
if (x == y) {
    document.write("x ist so groß wie y.");
}
```

Das folgende Beispiel testet mit dem Operator Modulo (%; vgl. Tabelle mit Operatoren), ob eine Zahl a durch eine Zahl b ohne Rest teilbar ist und gibt eine entsprechende Meldung in einem alert-Fenster aus:

```
  if (a % b == 0) {
    alert("a ist durch b ohne Rest teilbar!");
                        }
    else {
    alert("a ist nicht ohne Rest durch b teilbar!");
        }
```

10.6.2. *switch*-Anweisung

Um mehrere Alternativen zu prüfen, ist die if-else-Anweisung nicht optimal geeignet, da dann Verschachtelungen notwendig werden, die leicht unübersichtlich werden. Um dem zu begegnen, gibt es in JavaScript die Mehrfachverzweigung *switch*. Die Syntax der *switch*-Anweisung lautet:

```
switch (ausdruck) {
  case wert1:
    anweisung(en);
    break;
  case wert2:
    anweisung(en);
    break;
   ...
   default:
```

```
    anweisung(en);
 }
```

Mit switch leiten Sie die Fallunterscheidung ein (switch = Schalter). Dahinter folgt, in runde Klammern eingeschlossen, eine Variable oder ein Ausdruck, für dessen aktuellen Wert Fallunterscheidung durchgeführt werden sollen. Im folgenden Beispiel ist das die Variable mit dem Namen Eingabe. Diese Variable wird im Beispiel vor der Fallunterscheidung mit einem Wert belegt, den der Benutzer in einem Dialogfenster (window.prompt()) mit einem Eingabefeld eingibt. Der eingegebene Wert wird in der Variablen „Eingabe“ gespeichert. Die einzelnen Fälle, die unterschieden werden sollen, werden innerhalb geschweifter Klammern notiert und jeweils mit *case* eingeleitet. Dahinter folgt die Angabe des Wertes, auf den geprüft werden soll. Im Beispiel wird für jeden Fall eine Meldung ausgegeben. Wichtig ist dabei auch das Wort *break* am Ende jeder Switch-Anweisung. Falls man *break* weglassen würde, würden alle nachfolgenden Fälle auch ausgeführt, durch *break* wird die Switch-Anweisung verlassen. Für den Fall, dass keiner der definierten Fälle zutrifft, wird am Ende der Fall *default:* definiert, dessen Anweisungen ausgeführt werden, wenn keiner der anderen Fälle zutrifft.

```
var Eingabe = window.prompt("Geben Sie eine Ziffer zwischen 1 und 4
ein:","");
 switch(Eingabe) {
  case "1":  alert("Sie sind bescheiden!");
   break;
  case "2":  alert("Zwei ist mehr als eins!");
   break;
  case "3":  alert("Sie haben ein Dreirad gewonnen!");
   break;
  case "4":  alert("Nicht unbescheiden!");
   break;
  default:  alert("Bitte eine Ziffer zwischen 1 und 4 eingeben!");
   break; }
```

11. Funktionen

Funktionen finden sich als zentrales Konzept in vielen Programmiersprachen. Sie gewährleisten die Überschaubarkeit von komplexen Programmen und ermöglichen die Strukturierung von Programmen. Funktionen müssen einen dateiweit eindeutigen Namen besitzen. Für die Namensvergabe gelten die gleichen Regeln wie für die Vergabe von Variablennamen. Funktionen enthalten einen Block von Anweisungen. Im einfachsten Fall kann der Programmierer eine häufig benötigte Folge von Anweisungen als Funktion definieren und die Anweisungen an verschiedenen Stellen des JavaScript-Programms mit Hilfe des Funktionsnamens aufrufen und ausführen lassen. Das folgende Beispiel ist eine Funktion text(), die dazu dient einen häufig verwendeten String zu erzeugen:

```
function text()
    {
alert("JavaScript und immer wieder JavaScript!")
    }
```

Wenn diese Funktion so definiert ist, kann diese an späterer Stelle im Programm immer wieder aufgerufen werden, um den String "JavaScript und immer wieder JavaScript!" zu erzeugen.

Funktionen können auch ähnlich wie ihr mathematisches Pendant Parameter besitzen und einen Funktionswert in Abhängigkeit von diesen Parametern zurückliefern. Der grundlegende Aufbau einer Funktion stellt sich so dar:

```
function Name([parameter_1] [, parameter_2]) […, parameter_n])
    {
Anweisungsblock
return
    }
```

Die Definition einer JavaScript-Funktion wird durch das Schlüsselwort *function* eingeleitet. Danach folgt, durch ein Leerzeichen getrennt, der frei wählbare Funktionsnamen. Werden keine Parameter übergeben folgen dann eine öffnende und eine schließende Klammer. Parameter werden innerhalb der Klammer notiert, wobei mehrere Parameter jeweils durch ein Komma voneinander getrennt werden. Die Namen der Parameter sind frei wählbar. Diese Parameter werden *formale Parameter* genannt. Bei den Parameternamen gelten die gleichen Regeln wie beim Funktionsnamen. Mehrere Parameter werden durch Kommata voneinander getrennt. Die Anweisung *return* dient zum Zurückliefern von Werten an das aufrufende Programm (vgl. Beispiel für eine Funktion, die mit *return* einen Wert zurückliefert).

Beispiel für eine **Funktion mit einem Parameter**:

```
function PrimzahlCheck(Zahl)
 {
  var Grenzzahl = Zahl / 2;
  var Check = 1;
  for(i = 2; i <= Grenzzahl; i++)
    if(Zahl % i = = 0)
    {
     alert(Zahl + " ist keine Primzahl, weil teilbar durch " + i);
     Check = 0;
    }
  if(Check = = 1)
   alert(Zahl + " ist eine Primzahl!");
 }
```

Erläuterung:

Nach dem Schlüsselwort *function* folgt, durch ein Leerzeichen getrennt, ein frei wählbarer Funktionsname, im Beispiel "PrimzahlCheck". Innerhalb der folgenden Klammer steht der *formale Parameter* „Zahl“. Im Beispiel erwartet die Funktion PrimzahlCheck einen Parameter vom Typ Number. Der Wert des Parameters wird in der Funktion mehrmals benutzt. Dazu wird der Namen des Parameters verwendet (vgl. im Beispiel die fett formatierten Namen).

Der gesamte Inhalt der Funktion wird in geschweifte Klammern „{„ und „}“ eingeschlossen. Welche Anweisungen innerhalb einer Funktion stehen, hängt davon ab, was die Funktion leisten soll. Im obigen Beispiel wird ermittelt, ob die übergebene Zahl eine Primzahl ist. Wenn es keine ist, wird für jede Zahl, durch die sie teilbar ist, eine entsprechende Meldung ausgegeben. Wenn es eine Primzahl ist, wird am Ende ausgegeben, dass es sich um eine Primzahl handelt.

Eine Funktion wird mit ihrem Namen *aufgerufen* und im dem Namen folgenden Klammerpaar wird ggf. ein aktueller Parameter übergeben, der vom gleichen Typ wie der formale Parameter der Funktionsdeklaration sein muss. Z.B. kann die Funktion "PrimzahlCheck" im Programm aufgerufen werden mit PrimzahlCheck(8). Damit wird der aktuelle Parameter "8" an die Funktion übergeben und die Funktion gibt ein Meldungsfenster mit alert() aus, in dem gemeldet wird, dass 8 keine Primzahl ist.

Beispiel für eine **Funktion, die mit *return* einen Wert zurückliefert**:

```
function schreibeWortGross(Zeichenkette) {
Zeichen = Zeichenkette.toUpperCase();
return Zeichen;
}

var Wert = schreibeWortGross("Hallo")
document.write(Wert);
```

Um aus einer Funktion einen Wert an das aufrufende Programm zurückzuliefern, wird das Schlüsselwort *return* verwendet. Solche Rückgabewerte können vom Typ Boolean, Zahl oder String sein. Im obigen Beispiel wird der Variablen „Wert“ der Funktionsname „schreibeWortGross“ sowie als aktueller Parameter der String „Hallo“ übergeben. Innerhalb der Funktion wird lokal die Variable „Zeichen“ deklariert und dieser der formale Parameter „Zeichenkette“ sowie die Methode „toUpperCase()“ zugewiesen. Als Rück-

gabewert liefert die Funktion den aktuellen Inhalt der Variable „Zeichen“. Dieser ist die in Großbuchstaben umgewandelte Zeichenkette „HALLO“.

In JavaScript sind viele Funktionen integriert auf die man als Programmierer zurückgreifen kann. Bei diesen **integrierten Funktionen** handelt es sich nicht um Methoden, die fest an ein Objekt gebunden sind, wie z.B. die Methode alert(), der als Parameter ein String übergeben wird, der dann in einem Meldungsfenster ausgegeben wird, das mit einem OK-Button geschlossen werden kann. alert() ist eine Methode des Objektes „window“ (vgl. das folgende Kapitel zu Objekten). Integrierte Funktionen sind nicht an ein bestimmtes Objekt gebunden und können jederzeit aufgerufen werden.

decodeURI()	kodierten URI (Universal Resource Identifier, z.B. http://www.ph-freiburg.de) dekodieren
decodeURIComponent()	wie decodeURI(), aber mit dem Unterschied, dass auch folgende Zeichen dekodiert werden: , / ? : @ & = + $
encodeURI()	URI kodieren
encodeURIComponent()	wie encodeURI(), aber mit dem Unterschied, dass auch folgende Zeichen kodiert werden: , / ? : @ & = + $
eval()	interpretiert ein zu übergebendes Argument und gibt das Ergebnis zurück. Wenn das übergebene Argument als Rechenoperation interpretierbar ist, wird die Operation berechnet und das Ergebnis zurückgegeben; sehr praktisch, um als Zeichenketten notierte Rechenausdrücke mit einem einzigen Befehl errechnen zu lassen.
escape()	ASCII-Zeichen in Zahlen umwandeln
isFinite()	prüft, ob ein Wert sich innerhalb des Zahlenbereichs befindet, den JavaScript verarbeiten kann
isNaN()	auf numerischen Wert prüfen
parseFloat() ()	übergebene Zeichenkette in Kommazahl umwandeln
parseInt()	wandelt eine zu übergebende Zeichenkette in eine Ganzzahl (Integer) um und gibt diese als Ergebnis zurück
Number()	auf numerischen Wert prüfen
String()	in Zeichenkette umwandeln
unescape()	Zahlen in ASCII-Zeichen umwandeln

12. Exkurs Objektorientierte Programmierung (OOP)

12.1. Konzepte der objektorientierten Programmierung

In der folgenden Tabelle sind die wichtigen Konzepte der objektorientierten Programmierung (Datenkapselung, Vererbung, Polymorphie, Ereignissteuerung) und die zur Realisierung verwendeten Sprachstrukturen (Klassen oder Objektklassen, Klassenhierarchie, virtuelle Methoden, Versenden von Botschaften) aufgelistet:

Konzept	Sprachstruktur
DATENKAPSELUNG	*Klassen (Objektklassen)*
VERERBUNG	*Klassenhierarchie*
POLYMORPHIE	*virtuelle Methoden*
EREIGNISSTEUERUNG	*Versenden von Botschaften*

12.2. DATENKAPSELUNG und Klassen (Objektklassen)

Ein Ziel des objektorientierten Paradigmas ist es, den Prozess der Softwareentwicklung und den Programmierstil der menschlichen Denk- und Wahrnehmungsweise anzupassen. Dinge aus der Realität werden von Menschen in erster Linie danach beurteilt, was man mit ihnen machen kann, und in zweiter Linie danach, welche Eigenschaften sie haben. Fragt man Kinder oder Erwachsene, was denn ein Schraubenzieher sei, werden sie in der Regel zuerst erklären, dass man mit ihm Schrauben festziehen und lösen kann. Eigenschaften wie „länglich“ oder „spitz“ treten in den Vordergrund, wenn der Schraubenzieher als Hebel oder Stichwaffe zweckentfremdet wird. Menschen denken objektorientiert: Gegenstände werden unter dem Gesichtspunkt in Klassen eingeteilt, welche Operationen mit ihnen möglich sind und welche Eigenschaften sie besitzen.

Bei der prozeduralen Programmentwicklung werden Objekte in ihre Datentypen (z.B. ein Adressdatensatz (record)) und Prozeduren zum Umgang mit den Daten (Prozeduren zum Füllen, Lesen, Sortieren, ... der Datensätze) aufgeteilt. Im Vordergrund stehen dabei Überlegungen zu den Datentypen bzw. zu den Eigenschaften eines Objekts. Die Prozeduren zum Bearbeiten der Daten werden „später“ deklariert und realisiert. Der Programmierer muss dafür Sorge tragen, dass die Prozeduren auch auf die richtigen Daten zugreifen.

In der objektorientierten Programmierung werden dagegen die Daten (Felder) und Prozeduren in einer *Objektklasse* gekapselt (vgl. Abb.). Die Prozeduren einer solchen Klasse nennt man *Methoden*. Eine Objektklasse ist eine Softwarekomponente, die als eine Abstraktion eines „realen Dinges“ zu sehen ist. Solche Objekte haben einen inneren Zustand, der nur mit den zu diesem Objekt gehörigen Methoden manipuliert werden kann (*DATENKAPSELUNG*). Der Zustand eines Objekts wird durch seine Datenobjekte (Felder) und Eigenschaften (Properties, Attribute) repräsentiert. Solche Eigenschaften sind zum Beispiel die Eckpunktkoordinaten eines Rechtecks und seine Füllfarbe. Das Rechteck könnte Methoden zum Ändern seiner Größe, zum Verschieben und zum Ändern seiner Farbe haben. Die Werte der Eckpunktkoordinaten könnten nur von den Methoden des Rechtecks, z.B. den Methoden zum Ändern der Größe und zum Verschieben, verändert werden.

Objektorientierte Programmentwicklung lenkt die Sicht auf die Einheit der Methoden und der mit ihnen zu bearbeitenden Daten und versucht sich so den elementaren kognitiven Prozessen des Menschen beim Denken, Erkennen und Problemlösen anzupassen. Die Methoden eines Objekts können nur auf die Daten der zugehörigen Klasse zugreifen, und der Programmentwickler muss sich nicht darum kümmern, dass die Prozeduren mit den richtigen Daten arbeiten.

Wichtig ist die Unterscheidung einer *Objektklasse* von der *Objektinstanz*. Eine Objektklasse kann man sich als eine Schablone vorstellen nach deren Vorgaben ein reales Objekt erzeugt wird. Die Deklaration eines Formulars mit verschiedenen Eigenschaften, Datenfeldern (z.B. ein Button) und Methoden wäre eine

Objektklasse. Das Formularfenster, das zur Laufzeit am Bildschirm zu sehen ist, wäre dann die Objektinstanz.

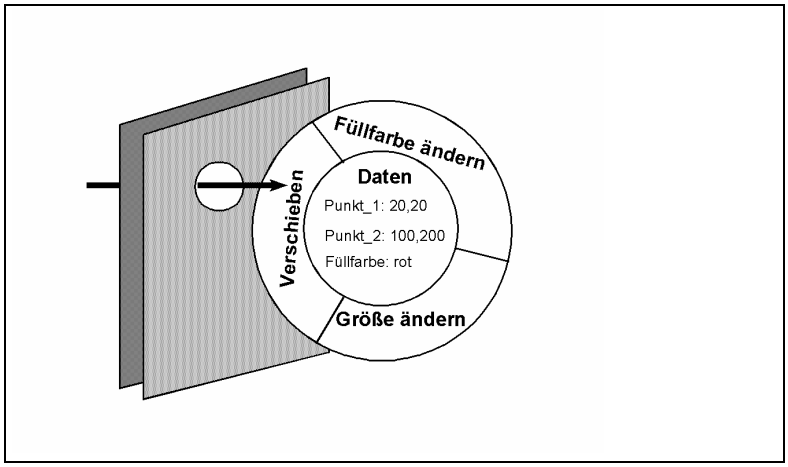


Abb.: Datenkapselung

12.3. VERERBUNG und Klassenhierarchie

Ein weiterer wesentlicher Aspekt objektorientierter Programmierung ist das Konzept der VERERBUNG. Jedes Objekt ist von einem bestimmten Typ, in der OOP *Klasse* genannt. Klassen legen für alle Objekte, die einer Klasse angehören, die gemeinsamen Datenfelder und Methoden, fest. Klassen werden aufgrund gemeinsamer Eigenschaften der Objekte gebildet. Trotzdem gibt es eine große Vielfalt von Klassen und es gibt sicher viele Methoden, die für mehrere dieser Klassen gelten könnten. Ein Ausweg aus diesem Dilemma ist das Verfahren der Taxonomie. Dabei werden die Klassen in eine Hierarchie gebracht, und Unterklassen sind in der Regel Spezialisierungen ihrer Oberklassen. Die Definition einer Unterklasse besteht dann im Hinzufügen weiterer und/oder im Redefinieren von geerbten Datenfeldern und Methoden. So erben die Unterklassen die Eigenschaften ihrer Oberklassen und werden um neue Datenfelder und Methoden ergänzt. Wichtig ist das Konzept der Vererbung für die Wiederverwendbarkeit von Software. Klassen können leicht bei anderen Projekten eingesetzt werden, gegebenenfalls ohne Änderung oder durch Bildung neuer Unterklassen.

12.4. POLYMORPHIE und virtuelle Methoden

In einer Hierarchie von Objektklassen erben die Nachkommen die Methoden ihrer Vorgänger. Falls das System eine Methode in der Klasse des aufrufenden Objekts nicht findet, steigt es automatisch in der Klassenhierarchie auf, bis es hier irgendwo auf sie trifft. Die gefundene Methode kommt dann zur Ausführung. Beispielsweise könnte ein Dialogfenster „Eingabedialog“ mit einem zusätzlichen Eingabefeld ein Nachkomme eines allgemeinen Dialogfensters „Dialogfenster“ mit OK- und einem Abbruch-Button sein, und das allgemeine Dialogfenster könnte ein Kind eines allgemeinen Fensters „Fenster“ sein. Jedes Fenster benötigt eine Methode „ZeigDich“. Diese Methode wird sinnvollerweise in der Klasse „Fenster“ vereinbart. Falls ein Dialogfenster „Eingabedialog“ die Aufforderung bekommt, sich zu zeigen, wird zuerst untersucht, ob dieses eine Methode „ZeigDich“ besitzt. Da dies nicht der Fall ist, wird beim Vorfahren „Dialogfenster“ gesucht. Da es auch hier keine passende Methode gibt, wird in der Hierarchie weiter gegangen und schließlich beim nächsten Vorfahren „Fenster“ die passende Methode gefunden. Diese kommt dann zur Ausführung. (vgl. Abb.)

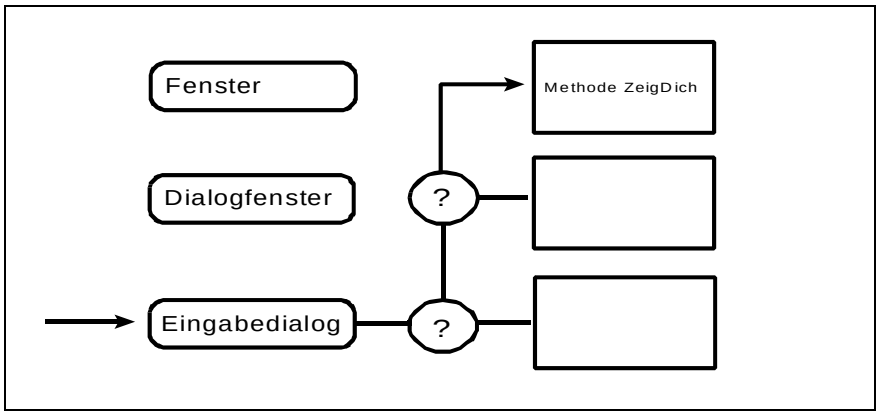


Abb.: Vererbung

Oftmals ist es aber so, dass Objekte eine Methode benötigen, die zwar prinzipiell die gleiche Aufgabe hat, aber je nach Klasse unterschiedlich implementiert werden sollte. Z.B. sollte die Methode „Zeichne“ in einer Objekthierarchie von geometrischen Figuren je nachdem, welches der Objekte Rechteck, Ellipse oder Kreis gezeichnet werden soll, unterschiedlich programmiert werden. Die Methode sollte aber immer „Zeichne“ heißen und nicht einmal „ZeichneRechteck“, einmal „ZeichneEllipse“ und das andere Mal „ZeichneKreis“. Es muss also eine Möglichkeit geben, eine einmal in einer Klasse implementierte Methode in Nachfolgerklassen zu überschreiben. Dies ist mit *virtuellen Methoden* möglich. Virtuelle Methoden ermöglichen die POLYMORPHIE (Vielgestaltigkeit) von Objekten (vgl. Abb.).

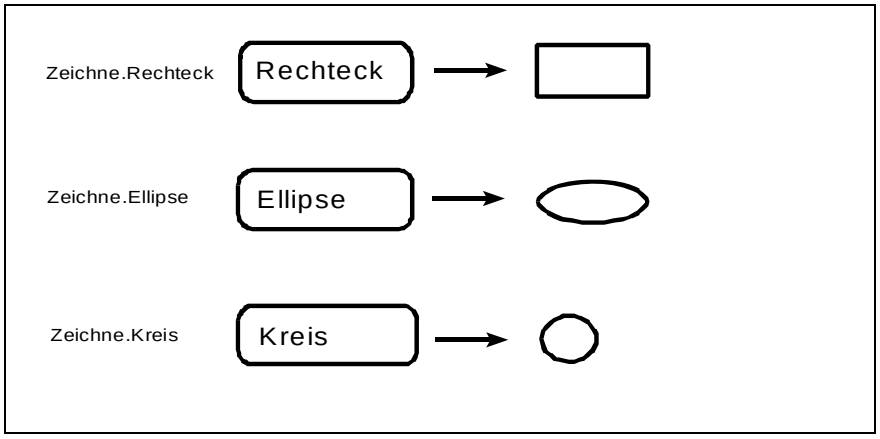


Abb.: Polymorphie

Virtuelle Methoden werden im Unterschied zu statischen Methoden dem Objekt nicht zum Zeitpunkt der Kompilierung (frühe Bindung) zugewiesen sondern erst während der Laufzeit (späte Bindung).

12.5. EREIGNISSTEUERUNG und Versenden von Botschaften

Die Dynamik einer objektorientierten Anwendung wird durch den Austausch von Nachrichten zwischen den Objekten realisiert. Ein Objekt *sendet eine Botschaft* mit einer bestimmten Anforderung. Diese Botschaft wird an den für sie bestimmten Empfänger weitergeleitet. Der Empfänger erzeugt mit einer seiner Methoden ein entsprechendes Resultat und sendet eine Botschaft mit dem Resultat an den Sender zurück.

Die Methoden einer Objektklasse legen fest auf welche Nachrichten das Objekt reagieren kann. Die Objekte sind selbst für die Ausführung einer Aufgabe verantwortlich - nicht das Programm. Objekte reagieren selbst auf Botschaften, die aufgrund bestimmter Ereignisse erzeugt wurden und ihm zugesandt wurden. OOP-Programme sind ereignisgesteuert (EREIGNISSTEUERUNG), nicht ablaufgesteuert wie „herkömmliche“ Programme (vgl. Abb.). Bei einem ablaufgesteuerten Programm sorgen Schleifen und Verzweigungen für die

Steuerung eines Programms, und es werden nur Eingaben akzeptiert, die dem aktuellen Modus des Programms entsprechen. Dabei steht das *Wie* der Ausführung einer Problemlösung im Vordergrund.

Bei der objektorientierten Programmentwicklung werden aufgrund bestimmter Ereignisse entstandene Botschaften, wie z.B. ein Mausklick des Benutzers an einer bestimmten Stelle auf dem Bildschirm an ein passendes Objekt weitergeleitet oder ein Objekt sendet selbst direkt eine Nachricht an ein anderes Objekt. Das Objekt, das die Nachricht empfängt, reagiert mit seinen Methoden auf die Botschaft. Der Programmierer teilt dabei lediglich mit, *was* erledigt werden soll und nicht *wie* es realisiert werden soll.

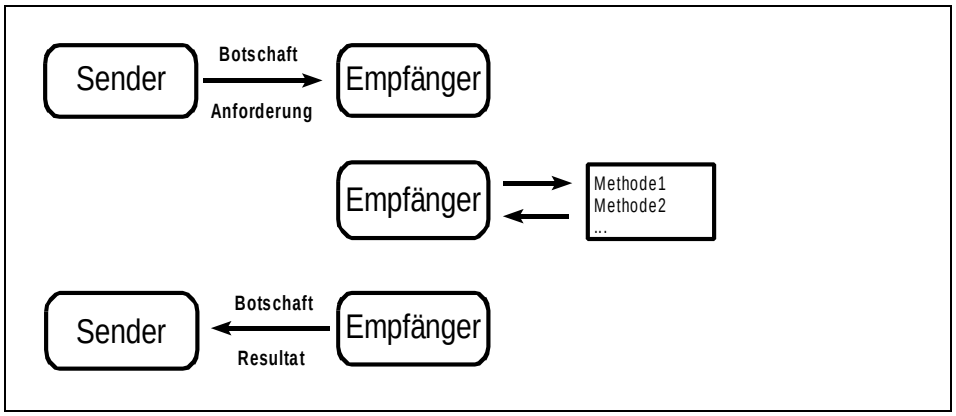


Abb.: Ereignissteuerung

13. Einige wichtige vordefinierte Objekte mit ihren Eigenschaften, Methoden und Unterobjekten – Document Object Model (DOM)

JavaScript lehnt sich an objektorientierte Programmiersprachen an. Mit JavaScript kann man zwar eigene Objekte definieren. Aber das Konzept der Vererbung steht nicht zur Verfügung. Ohne Vererbung macht natürlich auch Polymorhpie keinen Sinn. Daher bezeichnet man JavaScript auch nicht als objektorientierte, sondern als **objektbasierte** Sprache. JavaScrip-Programme sind aber auch ereignisgesteuert, wie wir eingangs gesehen haben.

Oft arbeitet man aber mit den in JavaScript schon vordefinierten Objekten.

Es gibt in JavaScript zwei Gruppen von **vordefinierten Objekten**, die sofort eingesetzt werden können.

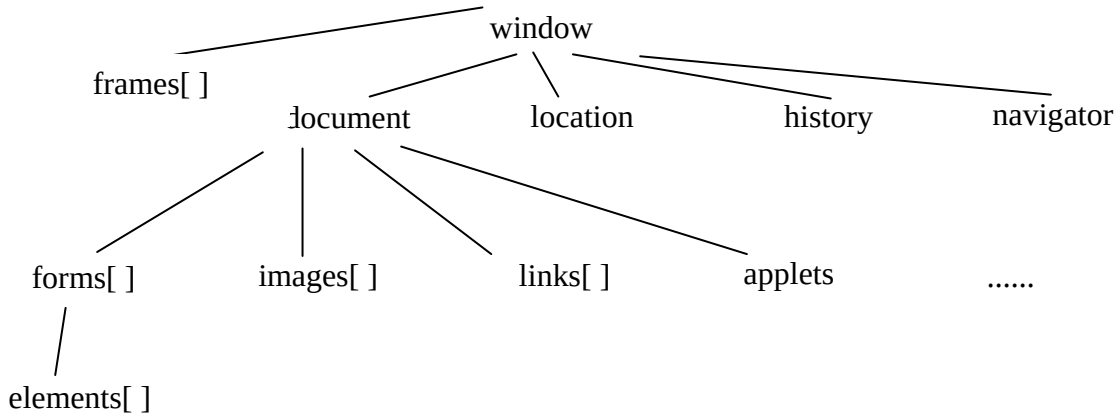
Die eine Gruppe von Objekten bezeichnet Netscape als **Kernobjekte** (core objects). Zu dieser Gruppe gehören Objekte für Zeit und Datum, mathematische Operationen, Zeichenmanipulationen, Felder (Arrays) zum Erzeugen von Serienvariablen, … Mit dem *Date*-Objekt können z.B. Datums- und Uhrzeitberechnungen durchgeführt werden. Die Eigenschaften des *Math*-Objekts sind bekannte mathematische Konstante, wie z.B. Pi. Die Methoden des Objekts erlauben beispielsweise trigonometrische Berechnungen, Runden oder die Ausgabe einer Zufallszahl.

Die zweite Gruppe hat das aktuelle Browser-Fenster als Ausgangspunkt. Diesem Objekt *window* sind die Objekte mit den Frames (*frames[ ]*) des Fensters, das Objekt *event*, der URL (*location*), die Historie der besuchten Seite (*history*), das Objekt *navigator* mit den Informationen zum Browser und ganz wichtig das Objekt *document* mit der HTML-Seite im Anzeigenbereich untergeordnet. Dem Objekt *document* sind die HTML-Tags definierte Elemente untergeordnet, wie zum Beispiel Formulare, Verweise, Grafikreferenzen usw. Für solche untergeordneten Elemente gibt es wieder eigene JavaScript-Objekte, zum Beispiel das Objekt forms für Formulare. Ein Formular besteht seinerseits aus verschiedenen Elementen, zum Beispiel aus Eingabefeldern, Auswahllisten oder Buttons zum Absenden bzw. Abbrechen. In JavaScript gibt es dafür ein Objekt *elements*, mit dem Sie einzelne Felder und andere Elemente innerhalb eines Formulars ansprechen können. Die Hierarchie der Objekte wurde vom W3-Konsortium im **Document Object Model (DOM)** als Reaktion auf die unterschiedliche JavaScript-Unterstützung beim Internet Explorer von Microsoft und Netscape Navigator 1998 in einer ersten Version festgelegt. Das W3-Konsortium erarbeitete dabei

jedoch keinen konkreten JavaScript-Standard, sondern ein. Dieses allgemeine Modell für Objekte eines Dokuments sollte eine Script-Sprache wie JavaScript, die sich als Ergänzungssprache zu Auszeichnungssprachen wie HTML versteht, vollständig umsetzen. 2000 wurde die Version 2.0 des DOM eine offizielle W3-Empfehlung, 2004 die Version 3.0. DOM wurde erstmals in der JavaScript-Version 1.5 umgesetzt.

Die Hierarchie der Objekte spiegelt in etwa die Hierarchie der HTML-Behälter (definiert durch die HTML-Tags) wieder:

window	gesamtes Browser-Fenster
document	<HTML> .... </HTML>
forms[ ]	<form> ... </form>
elements[ ]	<input>....</input>



Objekte haben Eigenschaften, wie z.B. das Objekt *window* einen Namen, eine Breite oder eine Höhe hat. Die **Eigenschaft** eines Objektes wird hinter dem Objektnamen getrennt durch einen Punkt notiert: *Objekt.Eigenschaft*. Z.B. kann auf die Eigenschaft *value* eines Formularelements mit dem Namen *Eingabe1* mit *Eingabe1.value* zugegriffen werden. Dabei ist zu beachten, dass nur Eigenschaften verwendet werden, die für das jeweilige Objekt definiert sind. Eigenschaften, die ein Objekt nicht kennt, führen zu Fehlermeldungen.

Außerdem können Objekte **Methoden** besitzen. Methoden sind Funktionen, die mit einer Klasse von Objekten verknüpft sind. Methoden können im Unterschied zu „normalen“ Funktionen nur im Zusammenhang mit einem bestimmten Objekt benutzt werden. Die Methoden sind an dieses Objekt gebunden. Die allgemeine Syntax für die Verwendung von Methoden lautet:
Objekt.Methode([Parameter1],[Parameter2], …])

Hinter dem Namen des Objekts wird durch einen Punkt getrennt die Methode notiert. Z.B. kann man mit der Methode *write* des Objekts *document* einen text im aktuellen Dokument ausgeben: *document.write(Hallo Welt!“)*. Es kann sein, dass Methoden keinen, einen oder mehrere Parameter erwarten. Es gibt aber auch Methoden bei denen die Parameter optional sind.

Man könnte untergeordnete Objekte auch als komplexe Eigenschaften der übergeordneten Objekte auffassen. Daher werden die einem Objekt untergeordneten Objekte auch genauso mit der Punktnotation angesprochen wie die Eigenschaften und Methoden eines Objekts.

Im Folgenden sind die Unterobjekte, Eigenschaften und Methoden der wichtigsten Komponenten einer WWW-Seite aufgelistet.

13.1. window

Unterobjekte:					
frames[ ] (Array von)Frames	document (HTML-Seite)	location (URL)	event (Ereignisse)	history (besuchte Seiten)	navigator (Browserinfos)

Eigenschaften	**Methoden**
closed (geschlossenes Fenster)	alert() (Dialogfenster mit Infos)
defaultStatus (Normalanzeige in der Statuszeile)	back() (zurück in History)
innerHeight (Höhe des Anzeigebereichs)	blur() (Fenster verlassen)
innerWidth (Breite des Anzeigebereichs)	captureEvents() (Ereignisse überwachen)
locationbar (URL-Adreßzeile)	clearInterval() (zeitliche Anweisungsfolge abbrechen)
menubar (Menüleiste)	clearTimeout() (Timeout abbrechen)
name (Fenstername)	close() (Fenster schließen)
outerHeight (Höhe des gesamten Fensters)	confirm() (Dialogfenster zum Bestätigen)
outerWidth (Breite des gesamten Fensters)	disableExternalCapture() (Fremdüberwachung verhindern)
pageXOffset (Fensterstartposition von links)	enableExternalCapture() (Fremdüberwachung erlauben)
pageYOffset (Fensterstartposition von oben)	find() (Text suchen)
personalbar (Zeile für Lieblingsadressen)	focus() (Fenster aktiv machen)
scrollbars (Scroll-Leisten)	forward() (vorwärts in History)
statusbar (Statuszeile)	handleEvent() (Ereignis übergeben)
status (Inhalt der Statuszeile)	home() (Startseite aufrufen)
toolbar (Werkzeugleiste)	moveBy() (bewegen mit relativen Angaben)
	moveTo() (bewegen mit absoluten Angaben)
	open() (neues Fenster öffnen)
	print() (ausdrucken)
	prompt() (Dialogfenster für Werteingabe)
	releaseEvents() (Ereignisse abschließen)
	resizeBy() (Größe verändern mit relativen Angaben)
	resizeTo() (Größe verändern mit absoluten Angaben)
	routeEvent() (Event-Handler-Hierarchie durchlaufen)
	scrollBy() (Scrollen um Anzahl Pixel)
	scrollTo() (Scrollen zu Position)
	setInterval() (zeitlische Anweisungsfolge setzen)
	setTimeout() (Timeout setzen)
	stop() (abbrechen)

Bemerkung:
Die Referenz auf das Objekt *window* kann weggelassen werden oder durch *this* ersetzt werden.

13.2. document

Unterobjekte:					
anchors[ ] (Verweisanker)	applets[ ] (Java-Applets)	embeds[ ] (Multimediaelemente)	forms[ ] (Formulare)	images[ ] (Grafiken)	layers[ ] (Layer)
links[ ] (Hyperlinks)	nodes[ ] (DOM-Knoten)	HTML-Elemente			

Eigenschaften	**Methoden**
alinkColor (Farbe für Verweise beim Anklicken)	captureEvents() (Ereignisse überwachen)

bgColor (Hintergrundfarbe) charset (verwendeter Zeichensatz) cookie (beim Anwender speicherbare Zeichenkette) defaultCharset (normaler Zeichensatz) fgColor (Farbe für Text) lastModified (letzte Änderung am Dokument) linkColor (Farbe für Verweise) referrer (zuvor besuchte Seite) title (Titel der Datei) URL (URL-Adresse der Datei) vlinkColor (Farbe für Verweise zu besuchten Zielen)	close() (schließen) createAttribute() (Attributknoten erzeugen) createElement() (Elementknoten erzeugen) createTextNode() (Textknoten erzeugen) getElementById() (HTML-Elementzugriff über id-Attribut) getElementsByName() (HTML-Elementzugriff über name-Attribut) getElementsByTagName() (HTML-Elementzugriff über Elementliste) getSelection() (selektierter Text) handleEvent() (Ereignisse verarbeiten) open() (Dokument öffnen) releaseEvents() (Ereignisse abschließen) routeEvent() (Event-Handler-Hierarchie durchlaufen) write() (ins Dokumentfenster schreiben) writeln() (zeilenweise schreiben)

13.3. forms

Unterobjekte:					
elements[ ] (Fomularelemente eines Formulars)	button (Button)	checkbox (Checkbox)	fileupload (Datei-Upload-Button)	hidden (Verstecktes Eingabefeld)	password (Passwortfeld)
radio (Radio-Button)	reset (Button zum Löschen)	select (Listenfeld)			

Eigenschaften	**Methoden**
action (Empfängeradresse bei Absenden) encoding (Kodierungstyp) length (Anzahl Formulare in Datei) method (Übertragungsmethode) name (Formularname) target (Zielfenster für CGI-Antworten)	handleEvent() (Ereignis verarbeiten) reset() (Formulareingaben löschen) submit() (Formular abschicken)

13.4. elements

Unterobjekte:					
options					

Eigenschaften	**Methoden**
checked (Angekreuzt) defaultChecked (vorangekreuzt) defaultValue (voreingegebener Wert) disabled (Element ausgrauen) form (Name des zugehörigen Formulars) length (Anzahl der Elemente eines Formulares) name (Elementname) type (Elementtyp) value (Elementwert/-inhalt)	blur() (Element verlassen) click() (Element anklicken) focus() (auf Element positionieren) handleEvent() (Ereginis verarbeiten) select() (Text selektieren)

Mit dem Objekt elements, das in der JavaScript-Objekthierarchie unterhalb des forms-Objekts liegt, haben Sie Zugriff auf Elemente eines Formulars.

13.5. select

Unterobjekt:	
options[ ] (Einträge des Listenfeldes)	

Eigenschaften	**Methoden**
defaultSelected (voreingestellte Auswahl) length (Anzahl der Auswahlmöglichkeiten) selected (aktuelle Auswahl) selectedIndex (Index der aktuellen Auswahl) text (Auswahltext) value (Auswahlwert)	

Das select-Objekt, welches eine Auswahlliste spezifiziert, kann ein Unterobjekt enthalten.

13.6. Ansprechen eines bestimmten Formularelementes

Schema 1	Beispiel 1
document.forms[#].elements[#].Eigenschaft document.forms[#].elements[#].Methode()	document.forms[0].elements[0].value = "Unsinn"; document.forms[0].elements[0].blur();

Schema 2	Beispiel 2
document.FormularName.Elementname.Eigenschaft document.FormularName.Elementname.Methode()	document.Testformular.Eingabe.value = "Unsinn"; document.Testformular.Eingabe.blur();

Formularelemente können Sie auf folgende Arten ansprechen:

- mit einer *Indexnummer* (wie in Schema 1 / Beispiel 1)
  Bei Verwendung von Indexnummern geben Sie *document.forms* an und dahinter in eckigen Klammern, das wievielte Formular im Dokument Sie meinen. Beachten Sie, dass der Zähler bei 0 beginnt, d.h. das erste Formular sprechen Sie mit *forms[0]* an, das zweite Formular mit *forms[1]* usw. Beim Zählen gilt die Reihenfolge, in der die <form>-Tags in der Datei notiert sind. Dann folgt die Angabe *elements*. Auch dahinter notieren Sie wieder eine Indexnummer in eckigen Klammern. Auch dabei wird wieder bei 0 zu zählen begonnen, d.h. das erste Element innerhalb eines Formulars hat die Indexnummer 0, das zweite die Indexnummer 1 usw.
- mit *Namen* (wie in Schema 2 / Beispiel 2)
  Dabei geben Sie mit *document.FormularName.ElementName* den Namen des Formulars und des Elements an, den Sie bei der Definition des Formulars und des Elements in den entsprechenden HTML-Tags im Attribut *name=* angegeben haben.

Man kann diese Zugriffsarten auch mischen und noch weitere Bezeichnungsarten finden. So ist z.B. auch folgendes möglich:

```
document.forms["Formular1"].elements[0].value
document.forms["Formular1"].Checkbox2.value
```

Beachten Sie:
Hinter dem, was hier als *elements* bezeichnet wird, verbergen sich **in Wirklichkeit mehrere, allerdings sehr ähnliche JavaScript-Objekte** (vgl. Tabelle mit Unterobjekten von forms**)** . So gibt es Objekte für

Klick-Buttons, Checkboxen, Datei-Buttons, Versteckte Elemente, Passwort-Felder, Radio-Buttons, Abbrechen-Buttons, Absenden-Buttons, einzeilige Eingabefelder und mehrzeilige Eingabefelder. Alle diese Objekte könne im "elements-Array" zusammengefasst werden. Alle diese Objekte werden auf die gleiche Art und Weise angesprochen. Sie unterscheiden sich lediglich in ihren Eigenschaften und Methoden.

14. Praxisbeispiele für JavaScript

14.1. Ausgabe von Text

1.Möglichkeit: Ausgabe in einem Meldungsfenster mit OK-Button

```
<script type="text/javascript">
   alert("Der Titel dieses Dokuments ist "+document.title);
 </script>
```

Der auszugebende Text darf ein beliebiger String-Ausdruck sein.

2.Möglichkeit: Ausgabe im Fenster, eingebettet in den restlichen HTML-Text

```
<script type="text/javascript">
   document.write("Der Titel dieses Dokuments ist "+document.title);
   document.write("<H1>Der Titel dieses Dokuments ist
   "+document.title+"</H1>");
   document.writeln("Der Titel dieses Dokuments ist "+document.title);
</script>
```

Der auszugebende Text darf ein beliebiger String-Ausdruck sein, der auch alle zugelassenen HTML-Tags enthalten kann. Ein auf diese Weise ausgegebener Text kann nur beim Laden oder erneuten Laden der Seite geschrieben werden. Für einen konstanten String hat *document.write* die gleiche Wirkung wie wenn der String unmittelbar in die HTML-Seite eingefügt worden wäre. Die Ausgabe mit *document.write* eröffnet aber die Möglichkeit, einen String auszugeben, der von Variablen Größen abhängt.

Die beiden oben genannten Möglichkeiten bieten sich für Kontrollausgaben zum Testen von JavaScript-Code an.

3. Möglichkeit: Ausgabe in einem Textfeld (Inputfeld) innerhalb eines Formulars

Bei dieser Möglichkeit kann die Ausgabe zu jeder Zeit erfolgen und kann auch verändert werden während die Seite geladen ist.

```
<FORM>
  <INPUT type="text" name="Testfeld">
  <INPUT type="button" value="Text ausgeben"
onclick="this.form.Testfeld.value='Hallo Welt' ">
</FORM>
```

14.2. Problem mit den Anführungszeichen

Bei folgender Zeile müssen die verschiedenen Anführungszeichen gemischt werden:

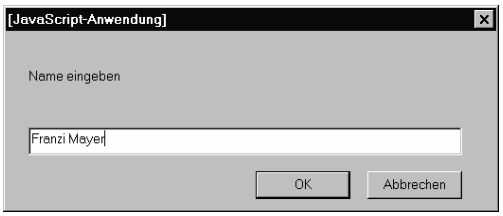
```
<input type=button onMouseDown="alert('Hallo Welt');">
```

Die äußeren Anführungszeichen müssen stehen, weil der JavaScript-Code für den Event-Handler in Anführungszeichen stehen muss, und die inneren Anführungszeichen stehen, weil *Hallo Welt* ein String sein soll. In diesem Fall müssen die zwei verschiedenen Typen von Anführungszeichen verwandt werden.

14.3. Eingabe von Text

1.Möglichkeit: Eingabe in einem Eingabefenster mit OK- und Abbrechen-Button

```
<script type="text/javascript" >
  var eingabe=prompt("Name eingeben");
  alert("Eingegeben wurde: "+eingabe);
</script>
```



Hier wird ein Text über ein Standard-Eingabefenster eingelesen und einer Variablen *eingabe* zugewiesen. Mit Hilfe der Variablen hat man weiter Zugriff auf die Eingabe.

2.Möglichkeit: Eingabe in einem Textfeld (Inputfeld) innerhalb eines Formulars

Bei dieser Möglichkeit kann die Ausgabe zu jeder Zeit erfolgen. Der Text kann auch verändert werden während die Seite geladen ist. Diese Möglichkeit zeigt das 3.Beispiel zur Ausgabe. In ein Input Feld mit Namen *Testfeld* wird Text eingegeben, auf den dann über *this.form.Testfeld.value* zugegriffen werden kann.

Sollen Zahlen eingegeben werden, dann werden diese zunächst als Text eingegeben und danach explizit mit einer der Umrechnungsfunktionen *parseInt()* oder *parseFloat()* umgewandelt werden, wenn JavaScript dies nicht schon implizit tut (vgl. lockere Typisierung von Variablen, S. 5f).

14.4. Grafikreferenzen auf Mausereignisse hin ändern

Beispiel:

```
<HEAD>
<TITLE>Ereignisbutton</TITLE>
<Script type="text/javascript">
  var Button1=new Image();
  var Button2=new Image();
  var Button3=new Image();

  Button1.src="komm.gif";
  Button2.src="drueck.gif";
  Button3.src="autsch.gif";

function wechsel (Quelle)
{
  window.document.images[0].src=Quelle;
}
</SCRIPT></HEAD>

<BODY LINK="WHITE" VLINK="WHITE" BGCOLOR="WHITE" TEXT="BLACK">
<H1 ALIGN="center">Testseite</H1>
<CENTER>
<A HREF="#" ONMOUSEOVER="wechsel(Button2.src)"
ONCLICK="wechsel(Button3.src)" ONMOUSEOUT="wechsel(Button1.src)" >
<IMG SRC="komm.gif"></A>
</CENTER>
</BODY></HTML>
```

Im Beispiel werden mit der Anweisung *var [ButtonName]=new Image();* 3 neue Bildobjekte angelegt. Den Objekten werden anschließend Grafikdateien als Quelle zugeordnet. Auf der HTML-Seite wird die Bilddatei „komm.gif“ dargestellt. Unter diesem Bild liegt ein Link, der verschiedene Mausereignisse entgegennehmen kann und die Funktion „wechsel“ aufruft, die dann den Verweis auf die Bilddateien ändert.

14.5. Neue Dokumente und Fenster öffnen

14.5.1. Neues Dokument im aktuellen Browserfenster öffnen

window.location.href=eingabe

```
<script type="text/javascript" >
   var eingabe=prompt("Name eingeben");
   window.location.href=eingabe; </script>
```

14.5.2. Neues Browserfenster mit neuem Dokument öffnen:

window.open("Auswahl1.htm")
window.open("Auswahl1.htm","Fenster1")

Öffnet ein neues Fenster. Erwartet mindestens zwei, optional auch drei Parameter:

1. URL-Adresse = Zieladresse einer Datei, die in das neue Fenster geladen werden soll. Wenn sich die Datei im gleichen Verzeichnis befindet, genügt der Dateiname. Ansonsten relative Pfadnamen oder absolute http-Adressen angeben. Bei Netscape darf dies auch eine leere Zeichenkette sein (öffnet ein leeres Fenster), was beim MS Internet Explorer allerdings zu einer Fehlermeldung führt.
2. Fenstername = Ein Name, der aus Buchstaben, Ziffern und Unterstrich bestehen darf. Unter diesem Namen können beispielsweise Verweise ihre Ziele mit <a href="ziel.htm" target="NameDesFensters"> in das erzeugte Fenster laden.
3. (optional) Angaben zum Aussehen des Fensters = Eine Zeichenkette, in der Sie die Größe und die Eigenschaften des Fensters festlegen können. Mehrere Angaben sind durch Kommata zu trennen.

Beispiel:

```
<html><head><title>Test</title>
  <script type="text/javascript">
  F1 =
open("dat1.htm","Fenster1","width=310,height=400,screenX=0,screenY=0");
  F2 =
open("dat2.htm","Fenster2","width=310,height=400,screenX=320,screenY=0");
  self.focus();
  self.close();
  </script>
  </head><body>
  </body></html>
```

Das Beispiel öffnet beim Einlesen der Datei zwei weitere Fenster so, dass sie nebeneinander angeordnet sind. Anschließend schließt sich das Hauptfenster selbst. Angenommen, in dat1.htm (wird ins erste Fenster geladen) steht ein Verweis, dessen Ziel im zweiten Fenster angezeigt werden soll. Dazu können Sie notieren:

```
<a href="dat3.htm" target="Fenster2">Verweistext</a>
```

Wichtig ist dabei die Angabe target=. Dort müssen Sie den Fenstername angeben, den Sie bei der Definition des gewünschten Zielfensters vergeben haben - im Beispiel "Fenster2".

Der Bezug *self* verweist auf das aktuelle Fenster, in dem dieses Dokument steht.

14.5.3. Dokument im aktuellen Browserfenster zum Neuschreiben öffnen

document.open()

Vorsicht: Diese Methode **öffnet nicht eine schon vorhandene Seite**, wie window.open()!
Soll eine vorhandene Seite ins aktuelle Fenster geladen werden muss man **window.location.href** ändern, wie oben beschrieben.

document.open() öffnet ein Dokument zum Schreiben. Dabei wird kein Fenster geöffnet, sondern der Fensterinhalt zum Neubeschreiben freigegeben. Falls das Dokument vorher einen Inhalt hatte, zum Beispiel eine zunächst geladene HTML-Datei, sollten Sie zuerst die *close()*-Methode aufrufen. Kann ohne, mit einem oder mit zwei Parametern aufgerufen werden. Folgende Parameter sind möglich:

1. Mime-Type = Eine Bezeichnung des Mime-Types für die Art der Daten, die in das Dokumentfenster geschrieben werden sollen. So können Sie das Dokumentfenster beispielsweise durch Angabe von "x-world/x-vrml" zum Schreiben von VRML-Code öffnen. Mit write()- und writeln()-Befehlen können Sie dann dynamisch VRML-Code schreiben.
2. replace = mit document.open("text/html","replace") öffnen Sie das Dokument zum Schreiben von HTML und bewirken, dass der neue Dokumentinhalt die gleiche Stelle in der History der besuchten Seiten einnimmt wie das Dokument, in dem der open-Befehl steht.

Beispiel:

```
<html><head><title>Test</title>
</head><body>
<script type="text/javascript">
function Wechsel()
{
 document.open();
 document.write("<a href=\"datei.htm\">Und jetzt steh ich hier</a>");
 document.close();
}
document.open();
document.write("<a href=\"javascript:Wechsel()\">Noch steh ich
hier</a>");
document.close();
</script>
</body></html>
```

Erläuterung:

Das Beispielscript führt zunächst den unteren Teil des Codes aus, da der obere in die Funktion *Wechsel()* eingebunden ist, die erst bei Aufruf ausgeführt wird. Im unteren Teil wird mit JavaScript dynamisch ein Verweis in das Dokumentfenster geschrieben. Wenn der Anwender den Verweis anklickt, wird die Funktion *Wechsel()* aufgerufen. Diese Funktion öffnet das Dokumentfenster zum neuen Schreiben und schreibt dynamisch einen Verweis auf die aktuelle Datei (im Beispiel wird angenommen, dass diese Datei *datei.htm* heißt). Klickt der Anwender auf den Verweis, wird die Datei erneut geladen, und dadurch wird wieder der erste Verweis geschrieben.

Beachten Sie, dass das Beispiel erst ab Netscape 3.x so funktioniert wie beschrieben. Netscape 2.x kennt die *document.open()* zwar auch schon, doch leert dabei das Dokumentfenster nicht.

14.6. Formulare: Verarbeitung der eingegebenen Daten

14.6.1. Verarbeitung auf der gleichen Seite wie das Formular

Die in ein Formular eingegebenen Daten können mit JavaScript auf der gleichen Seite verarbeitet werden auf der das Formular steht. Beispiele dafür haben Sie schon mehrfach gesehen. Ergebnisse müssen dann in Input- oder Textarea-Feldern ausgegeben werden. Sollen die Ergebnisse auf einer anderen Seite als der aktuellen erscheinen, so kann man entweder im selben Fenster ein neues Dokument erzeugen und dieses neu schreiben oder ein neues Fenster mit einem neuen Dokument erzeugen (→ Neue Dokumente und Fenster öffnen).

14.6.2. Verarbeitung auf einer anderen Seite

```
<form name="Formular1" action="testseite.htm" method =get>
    ...
<input type=submit  name="Abschicken" value="Abschicken">
</form>
```

Seite mit Formular mit Verarbeitung durch eine andere Seite

```
<HTML>
<HEAD>
   <TITLE>Objekte-test_andereSeite</TITLE>
</HEAD>
<BODY>
<form name="Formular1" action="testseite.htm" method =get> <!—muss get-
Methode verwenden →
Checkbox
<BR>
<input type=checkbox  name="Checkbox1" checked
value="cwert1">Auswahl1</input><BR>
<input type=checkbox  name="Checkbox2" checked
value="cwert2">Auswahl2</input><BR>
<input type=checkbox  name="Checkbox3" value="cwert3"
>Auswahl3</input><BR>
<input type=checkbox  name="Checkbox4"
value="cwert4">Auswahl4</input><BR><BR>
<input type=submit  name="Abschicken" value="Abschicken"></input><BR><BR>
</form>
</BODY>
</HTML>
```

Testseite dazu

```
<HTML>
<HEAD>
   <TITLE>testseite</TITLE>
</HEAD>
<BODY>
 Seite zum Testen von „Submit an Seiten" (Datei Objekte-
test_andereSeite.htm)
 <BR><BR><BR>
 <SCRIPT language=javascript>
 document.writeln(document.location);   // hier könnte die Verarbeitung
geschehen
 </SCRIPT>
</BODY>
</HTML>
```

14.7. Eingaben an eine E-Mail-Adresse schicken

Statt die Eingaben in ein Formular automatisch verarbeiten zu lassen kann man die Einträge auch an eine E-Mail-Adresse schicken lassen und dort von Hand weiter bearbeiten (oder von einem Mail-Hilfsprogramm verarbeiten lassen).

```
<form name="Formular1" action="mailto:deissler@ruf.uni-freiburg.de"
method =post>
    ...
<input type=submit  name="Abschicken" value="Abschicken">
</form>
```

Seite mit Formular mit Verarbeitung durch eine andere Seite

```
<HTML>
<HEAD>
   <TITLE>Objekte-test_mailto</TITLE>
</HEAD>
<BODY>
<form name="Formular1" action="mailto:deissler@ruf.uni-freiburg.de"
method =post>
<!-- muss post-Methode verwenden -->
Checkbox
<BR>
<input type=checkbox  name="Checkbox1" checked
value="cwert1">Auswahl1</input><BR>
<input type=checkbox  name="Checkbox2" checked
value="cwert2">Auswahl2</input><BR>
<input type=checkbox  name="Checkbox3" value="cwert3"
>Auswahl3</input><BR>
<input type=checkbox  name="Checkbox4"
value="cwert4">Auswahl4</input><BR>
<input type=submit  name="Abschicken" value="Abschicken"></input><BR>
</form>
</script>
</BODY>
</HTML>
```

14.8. Eingaben an einen Server schicken und mit einem CGI-Script verarbeiten

Diese Möglichkeit ist am schwierigsten, wird aber am häufigsten benötigt. Das Problem besteht darin, dass der Client die Berechtigung haben muss, auf dem Server-Rechner eigene Programme zu installieren. Darüber hinaus erfordert diese Verarbeitung Kenntnisse einer weitergehenden Programmiersprache oder den Einsatz entsprechender Software.

Im folgenden Beispiel wird ein Script *environ.pl* in der Sprache Perl aufgerufen, das nur die Werte aus dem Formular zeigt.

```
<HTML>
<HEAD>
<TITLE>Testseite</TITLE>
</HEAD>
<BODY>
<H1>Testseite mit GET-Verfahren</H1>
<FORM method="get" action="http://127.0.0.1/cgi-bin/environ.pl">
Das Input-Feld:  
<INPUT type="TEXT"   NAME="botschaft">
```

```
<INPUT type="SUBMIT"  VALUE="Botschaft absenden"></FORM>
<P>Die Testseite <B>mu&szlig;</B> im Hauptverzeichnis des Servers stehen.
<BR>Hier wird ein Perl-Script aufgerufen, das alle Umgebnungsvariablen
zeigt.
<BR> 
</BODY>
</HTML>
```

15. Übersicht über die Sprachelemente von JavaScript

!= (Vergleichsoperator)
$[1..9] / RegExp
% (Berechnungsoperator)
&& (Logischer Operator)
* (Berechnungsoperator)
+ (Berechnungsoperator)
+ (Zeichenkettenoperator)
++ (Berechnungsoperator)
- (Berechnungsoperator)
-- (Berechnungsoperator)
/ (Berechnungsoperator)
: (Entweder-Oder-Abfrage)
< (Vergleichsoperator)
<= (Vergleichsoperator)
= (Zuweisungsoperator)
== (Vergleichsoperator)
> (Vergleichsoperator)
>= (Vergleichsoperator)
? (Entweder-Oder-Abfrage)
|| (Logischer Operator)
abs() / Math
above / layers
acos() / Math
action / forms
alert() / window
alinkColor / document
all (Objekt)
altKey / event
anchors (Objekt)
appCodeName / navigator
applets (Objekt)
appName / navigator
appVersion / navigator
arguments / Function
arity / Function
asin() / Math
atan() / Math
availHeight / screen
availWidth / screen
back() / history
back() / window
background / layers
below / layers
bgColor / document
bgColor / layers
big() / string
blink() / string
blur() / elements
blur() / window
bold() / string
border / images
Boolean (Objekt)
break
caller / Function
captureEvents() / document
captureEvents() / layers
captureEvents() / window
case
ceil() / Math
charAt() / string
charCodeAt() / string
charset / document
checked / elements
className / style
clearInterval() / window
clearTimeout() / window
click() / elements
click() / all
clientX / event
clientY / event
clip / layers
close() / document
close() / window
closed / window
colorDepth / screen
concat() / Array
concat() / string
cookie / document
complete / images
confirm() / window
contains() / all
continue
cos() / Math
ctrlKey / event
dataFld / all
dataFormatAs / all
dataPageSize / all
dataSrc / all
Date (Objekt)
default
defaultCharset / document
defaultChecked / elements
defaultSelected / options
defaultStatus / window
defaultValue / elements
description / mimeTypes
description / plugins
disableExternalCapture() / window
do
document (Objekt)
document / layers
E / Math
elements (Objekt)
else
enabledPlugin / mimeTypes
enableExternalCapture() / window
encoding / forms
escape()
eval()
event (Objekt)
exec() / RegExp
exp() / Math
getSelection() / document
fgColor / document
filename / plugins
find() / window
fixed() / string
floor() / Math
focus() / elements
focus() / window
fontcolor() / string

fontsize() / string
for
for in
form / elements
forms (Objekt)
forward() / history
forward() / window
fromCharCode() / string
function
Function (Objekt)
getAttribute() / all
getAttribute() / style
getDate() / Date
getDay() / Date
getHours() / Date
getMinutes() / Date
getMonth() / Date
getSeconds() / Date
getTime() / Date
getTimezoneOffset() / Date
getYear() / Date
go() / history
handleEvent() / document
handleEvent() / elements
handleEvent() / forms
handleEvent() / images
handleEvent() / layers
handleEvent() / window
hash / location
height / images
height / screen
history (Objekt)
home() / window
host / location
hostname / location
href / location
hspace / images
id / all
if
images (Objekt)
indexOf() / string
innerHeight / window
innerHTML / all
innerText / all
innerWidth / window
insertAdjacentHTML() / all
insertAdjacentText() / all
isNaN()
isTextEdit / all
italics() / string
javaEnabled / navigator
javascript:
join() / Array
keyCode / event
lang / all
language / all
language / navigator
lastIndexOf() / string
lastModified / document
layers (Objekt)
layerX / event
layerY / event
left / layers
length / all
length / anchors
length / applets
length / Array
length / forms
length / history
length / images
length / layers
length / links
length / mimeTypes
length / options
length / plugins
length / string
link() / string
links (Objekt)
linkColor / document
LN2 / Math
LN10 / Math
load() / layers
location (Objekt)
locationbar / window
log() / Math
LOG2E / Math
LOG10E / Math
lowsrc / images
Math (Objekt)
match() / string
max() / Math
MAX_VALUE / Number
menubar / window
method / forms
mimeTypes (Objekt)
min() / Math
MIN_VALUE / Number
modifiers / event
moveAbove() / layers
moveBelow() / layers
moveBy() / layers
moveBy() / window
moveTo() / layers
moveTo() / window
moveToAbsolute() / layers
new
name / elements
name / forms
name / images
name / layers
name / plugins
name / window-Objekt
NaN / Number
navigator (Objekt)
NEGATIVE_INFINITY / Number
Number (Objekt)
Number()
offsetHeight / all
offsetLeft / all
offsetParent / all
offsetTop / all
offsetWidth / all
offsetX / event
offsetY / event
onAbort
onBlur
onChange
onClick
onDblClick
onError
onFocus
onKeydown
onKeypress
onKeyup
onLoad
onMousedown
onMousemove
onMouseout
onMouseover
onMouseup
onReset
onSelect
onSubmit
onUnload
open() / document
open() / window

opener / window
options (Objekt)
outerHeight / window
outerHTML / all
outerText / all
outerWidth / window
pageX / layers
pageX / event
pageXOffset / window
pageY / layers
pageY / event
pageYOffset / window
parentElement / all
parentLayer / layers
parentTextEdit / all
parse() / Date
parseFloat()
parseInt()
pathname / location
recordNumber / all
personalbar / window
PI / Math
pixelDepth / screen
platform / navigator
plugins (Objekt)
pop() / Array
port / location
POSITIVE_INFINITY / Number
pow() / Math
print() / window
prompt() / window
protocol / location
push() / Array
random() / Math
referrer / document
RegExp (Objekt)
releaseEvents() / document
releaseEvents() / layers
releaseEvents() / window
reload() / location
removeAttribute() / all
removeAttribute() / style
replace() / location
replace() / string
reset() / forms
resizeBy() / layers
resizeBy() / window
resizeTo() / layers
resizeTo() / window
return
reverse() / Array
round() / Math
routeEvent() / document
routeEvent() / layers
routeEvent() / window
screen (Objekt)
screenX / event
screenY / event
scrollbars / window
scrollBy() / window
scrollIntoView() / all
scrollTo() / window
search / location
select() / elements
selected / options
selectedIndex / options
self / window
setAttribute() / all
setAttribute() / style
setDate() / Date
setHours() / Date
setInterval() / Date
setMinutes() / Date
setSeconds() / Date
setTime() / Date
setTimeout() / window
setYear() / Date
shift() / Array
shiftKey / event
siblingAbove / layers
siblingBelow / layers
sin() / Math
slice() / Array
slice() / string
small() / string
sourceIndex / all
splice() / Array
split() / string
sort() / Array
sqrt() / Math
SQRT1_2 / Math
SQRT2 / Math
src / images
src / layers
statusbar / window
status / window
stop() / window
strike() / string
string (Objekt)
style (Objekt)
sub() / string
submit() / forms
substr() / string
substring() / string
suffixes / mimeTypes
sup() / string
switch
tagName / all
tags / all
tan() / Math
target / forms
test() / RegExp
text / options
this
title / document
title / all
toGMTString() / Date
toLocaleString() / Date
toLowerCase() / string
top / layers
toUpperCase() / string
toolbar / window
type / elements
type / event
type / mimeTypes
unescape()
unshift()
userAgent / navigator
URL / document
UTC() / Date
var
value / elements
value / options
vlinkColor / document
visibility / layers
vspace / images
which / event
while
width / images
width / screen
window (Objekt)
with()
write()
writeln()
x / event
y / event

zIndex / layers