

Thomas Feuerstack
Beratung und IT-Services



JavaScript in 2 Tagen

Inhaltsverzeichnis

1	Die Vorstellungsrunde	1
1.1	Das sind Sie	1
1.2	Das ist JavaScript	2
1.3	Das kann JavaScript nicht	3
I	Erster Tag	5
2	Hallo Welt	7
2.1	Die Arbeitsumgebung	7
2.2	Das erste Lebenszeichen	7
2.3	Externes JavaScript	10
2.4	Debugging	11
3	Basiswissen	13
3.1	Variablen	13
3.2	Arrays	17
3.3	Kommentare	19
3.4	Zuweisungen	20
3.5	Arithmetische Operationen	21
3.6	Mathematische Funktionen	22
3.7	Funktionen für Zeichenketten	22
3.8	Einfache Ein- und Ausgaben	26
3.9	Übungsaufgaben	29
4	Kontrollstrukturen	31
4.1	Verzweigungen und Fallunterscheidungen	32
4.2	Schleifen	35
4.3	Übungsaufgaben	38
4.4	Funktionen	39

4.5	Variablen und ihr Gültigkeitsbereich	42
4.6	Übungsaufgabe	46
II	Zweiter Tag	47
5	JavaScript und die HTML-Umgebung	49
5.1	Ereignisse und Events	49
5.2	Weitere Dokumente laden	55
5.3	Eigene Fenster aufbauen	56
5.4	HTML-Objekte	58
5.5	Übungsaufgaben	65
5.6	Interaktion mit dem WWW-Browser	66
5.7	Übungsaufgaben	69
6	Erweiterte Techniken	71
6.1	Zeitgeber	71
6.2	Übungsaufgabe	74
7	Für alle anderen Tage...	75
7.1	Nie, Nie, ... Niemals!	75
A	Lösungen aller Aufgaben	77
I	Index	87

1 Die Vorstellungsrunde

1.1 Das sind Sie

Eigentlich gibt es heutzutage kein schriftliches Werk mehr, in dem nicht mindestens fünf Kapitel zur sogenannten „Einführung“ gehören würden. Je nachdem an welchen Kundenkreis, bzw. deren bereits vorhandene Sachkenntnisse, sich der Autor wendet, wimmelt es in der Einführung von guten Ratschlägen. Beispielhaft sei hier erwähnt, dass Tastaturbefehle und Bildschirminhalte normalerweise in einer anderen Schriftart dargestellt werden, und dass eine Maus in aller Regel zwei Tasten besitzt.¹ Nebenbei sollte nicht vergessen werden, dass ein Kapitel mit der Überschrift **Übungsaufgaben** Anregungen für den Aufbau eines eigenen Erfahrungsschatzes enthält.

Ich gehe mal davon aus, dass ich mir solche Instruktionen bei Ihnen sparen kann. Sie sollten (nicht zuletzt aufgrund des Layouts) auf den folgenden Seiten selbst unterscheiden können, an welchen Stellen es sich um erläuternde Hinweise, und wo es sich um Beispiele von JavaScript-Code handelt. Ganz nebenbei hoffe ich, dass Sie sich im Umgang mit Personal Computern allgemein auskennen. Sollten Ihnen auch noch Begriffe wie „Editor“ und „Web-Browser“ geläufig sein, liegen wir schon im Großen und Ganzen auf einer gemeinsamen Linie. Für den Fall, dass die beiden zuletzt vorgestellten Begriffe hingegen gar keine Bedeutung für Sie haben, legen Sie dieses Skript bitte sofort zur Seite – es wird Ihnen mit Sicherheit keine Freude bereiten.

- ! → Sie lesen noch weiter? Gut, denn dann sollten Sie noch unbedingt eine weitere Voraussetzung erfüllen. Sie sollten in jedem Fall (zumindest rudimentäre) HTML-*Kenntnisse* haben, weil HTML nun einmal die primäre Beschreibungssprache von Web-Seiten ist, und weil JavaScript-Code nun einmal in HTML-Seiten untergebracht wird. Das Andere ohne das Eine zu wollen wäre daher gleichbedeutend mit dem Umstand eine DVD zu kaufen, ohne ein Abspielgerät zu besitzen.

Bevor Sie das Skript jetzt weglegen: An der gleichen Stelle, an der Sie diese JavaScript-Einführung gefunden haben, existiert unter der Signatur A/005/9802 die Broschüre „[Die kleine HTML-Schule](#)“^[2] mit deren Hilfe Sie eventuelle Defizite rasch ausgleichen können.

Weiterhin ist es von Vorteil, wenn auch nicht unbedingt Voraussetzung, wenn Sie bereits Programmierkenntnisse besitzen. Sie können durch dieses Skript einen Einstieg in JavaScript und damit in eine Programmiersprache finden, dies ist aber nicht unbedingt gleichbedeutend mit: „Sie

¹ die sich darüberhinaus auch noch bei den meisten Betriebssystemen benutzen lassen.

werden durch dieses Skript Programmieren lernen.“ Zum Programmieren brauchen Sie einen Blick, wie unterschiedlichste Problematiken, beziehungsweise deren Lösung, durch Programmkonstrukte abgebildet werden können. Halten Sie sich das landläufige Beispiel vor Augen, wonach man Autofahren auch erst *nach* der Fahrschule lernt.

In Analogie zum oberen Beispiel braucht man für dieses „Programmiergefühl“ daher auch mehr Übung und Erfahrung, als Ihnen diese Broschüre so ohne Weiteres vermitteln könnte. Sie sollten die darin enthaltenen Beispiele und Aufgaben dazu nutzen, um Ihren Blick in diese Richtung zu schärfen.

Vielleicht machen wir jetzt schnell weiter, bevor gar keiner mehr mitliest.

1.2 Das ist JavaScript

JavaScript ist eine interpretierte Programmiersprache mit rudimentärer Objektorientierung – so erzählt es uns das (ansonsten ausgezeichnete) Buch „JavaScript, Das umfassende Referenzwerk“^[1]. Nicht, dass in diesem Satz auch nur ansatzweise die Unwahrheit vertreten wäre, es ist im Gegenteil die denkbar knappste Definition, die man über JavaScript geben kann. Ich vermute aber, es ist Ihnen ähnlich wie mir gegangen, als ich den Satz das erste Mal gelesen habe: Ich habe ihn nicht verstanden. Lassen Sie uns daher einfach Punkt für Punkt durchgehen, woher wir JavaScript kennen, um herauszufindenzukriegen was JavaScript nun ist.

Den meisten von uns wird JavaScript in Zusammenhang mit WWW-Seiten aufgefallen sein. Bekanntermassen werden Inhalte im World Wide Web über die Seitenbeschreibungssprache HTML dargestellt. HTML selbst hat aber nun einen entscheidenden Nachteil: Es ist eine sehr „starre“ Angelegenheit in dem Sinne, dass einmal auf dem Bildschirm sichtbare Elemente nicht mehr beeinflusst werden können. Diese Beeinflussungen oder gar Veränderungen können jedoch durch eingebetteten JavaScript-Code vorgenommen werden, wodurch auf einmal Bewegung in die ganze Angelegenheit kommt. Generell lässt sich der Begriff Beeinflussungen mit den folgenden Begriffen beschreiben:

Steuerung von Aussehen und Inhalt von (WWW-)Dokumenten : Mit Hilfe von JavaScript ist es möglich, beliebigen Text in HTML-Dokumente zu schreiben, beispielsweise ein aktualisiertes Tagesdatum. Darüberhinaus können Dokumentressourcen wie Farben, Schriftgrößen, usw. durch JavaScript geändert werden.

Steuerung des Web-Browsers : Durch JavaScript können Sie den WWW-Browser veranlassen weitere Fenster und Dialogboxen zu öffnen, Frameinhalte zu beeinflussen und die browsereigene Historie zu verarbeiten.

Interaktion mit dem Dokumentinhalt : Prinzipiell ist es schon seit längerer Zeit möglich, beispielsweise Bildinhalte oder Formulare einer WWW-Seite mit Hilfe von JavaScript zu manipulieren. Unterstützt Ihr WWW-Browser das

Document Object Model (DOM), so ist der Zugriff auf beinahe alle HTML-Marken möglich.

Soweit alles klar? Stellen Sie sich dann für den Moment die Programmiersprache JavaScript als die übliche Untermenge der englischen Sprache vor, so wird klar, dass es ein weiteres Programm geben muss, welche den JavaScript-Englisch-Slang in für Ihren WWW-Browser ausführbare Anweisungen umsetzt; einen sogenannten *Interpreter*. Sie erinnern sich? „JavaScript ist eine *interpretierte* Programmiersprache...“

Woher Sie diesen Interpreter nehmen, braucht Sie vorerst nicht zu kümmern, er wird von den meisten WWW-Browsern automatisch bereit gestellt.

1.3 Das kann JavaScript nicht

Nicht zuletzt durch die oben vorgestellten Eigenschaften stehen viele WWW-Autoren JavaScript auch kritisch gegenüber, schließlich kann jedes sinnvolle Feature auch missbräuchlich eingesetzt werden. Die folgenden Eigenschaften sind aus diesem Grund *nicht* in JavaScript enthalten:

- JavaScript besitzt keinen Zugriff auf Dateien innerhalb Ihres lokalen Dateisystems – ich denke, das würden Sie auch nicht wirklich wollen.
- JavaScript unterstützt keinerlei Netzwerk-Operationen, mit der Ausnahme: Es kann einen WWW-Browser dazu veranlassen, beliebige URLs vom Netz zu laden.

Bei der Gelegenheit sollte ich gleich noch einen weiteren, häufig gehörten Irrtum aufklären. JavaScript besitzt keine verwandtschaftlichen Verwandlungen zu der ähnlich benannten, objektorientierten Programmiersprache Java – es ist schon gar keine vereinfachte Untermenge davon. Ursprünglich sollte JavaScript unter dem Namen LiveScript veröffentlicht werden; der Name wurde erst im letzten Moment aus Marketinggründen geändert.²

Ganz nebenbei ist JavaScript aber auch keine „einfache“ Programmiersprache, aber da werden Sie aber der nächsten Seite schnell von selbst hintersteigen.

² Sofern Sie das nicht verstehen, denken Sie mal darüber nach, warum Deutschlands Urwaschmittel *Persil* heisst, und die später folgenden Kokurrenzprodukte Namen wie *Sunil*, *Saptil* und *Tandil* haben.

I Erster Tag

2 Hallo Welt

Alle diejenigen, die schon einmal ein Buch in der Hand gehalten haben das sich mit dem Erlernen einer Programmiersprache befasst, ahnen wahrscheinlich längst was jetzt kommt: das unvermeidliche allererste Programm (das eigentlich eher ein Progrämmchen ist).

Seit Kernighan/Ritchies Buch „Programmieren in C“^[3] erblicken alle programmierten Erstlinge mit den Worten „Hallo Welt“ das Licht derselben. Eine Änderung dieses Zustands wäre gleichbedeutend mit dem Untergang aller Werte des christlichen Abendlandes.

Egal aber ob Sie das christliche Abendland zukünftig stürzen oder stützen¹ wollen, Sie benötigen in jedem Fall das passende Werkzeug für dieses Unterfangen.

2.1 Die Arbeitsumgebung

Analog zur Erstellung von HTML-Dateien müssen Sie für JavaScript ausgesprochen wenig investieren um durchzustarten. Es genügen:

1. Ein handelsüblicher Texteditor wie Emacs, vi oder Kate (alle für UNIX), bzw. Windows Standardeditor *Notepad* oder der komfortablere (und frei erhältliche) *Syn*. Selbst die Verwendung der vermeintlichen Allzweckwaffe *Word* ist denkbar, sofern Sie beim Abspeichern darauf achten, dass das erfasste Dokument als ASCII-Text abgelegt wird.
2. Ein allgemein verbreiteter WWW-Browser wie Microsofts Internet Explorer, Suns Netscape oder alternative freie Entwicklungen wie Mozilla und Opera. Hauptsache, es ist ein JavaScript-Interpreter darin enthalten.

Wie bei jeder interpretierten Programmiersprache agieren die Interpreter der verschiedenen Hersteller unterschiedlich. Speziell am Anfang, an dem wir uns ja gerade befinden, sollte es jedoch zu keinen gravierenden Abweichungen kommen.

2.2 Das erste Lebenszeichen

Nachdem Sie alle benötigten Arbeitsmittel zusammengeklaut haben, starten Sie den bereits erwähnten handelsüblichen Texteditor, und erfassen die folgenden Zeilen:

¹ Spätestens an dieser Stelle sollten Sie die Macht eines einzelnen Buchstaben erkannt haben.

```
1 <html>
2 <body>
3   <p>Vor Aufruf des JavaScripts</p>
4   <script language="javascript">
5     <!--
6     // Jede Menge JavaScript-Anweisungen
7     alert("Hallo Welt!");
8     window.document.write("<p>Innerhalb des JavaScripts</p>");
9     //-->
10  </script>
11  <p>Nach Aufruf des JavaScripts</p>
12 </body>
13 </html>
```

Speichern Sie den Inhalt unter dem Namen `HalloWelt.html`,² und öffnen Sie diese Datei in einem beliebigen Web-Browser. Abbildung 2.1 sollte Ihrem Ergebnis weitestgehend entsprechen.

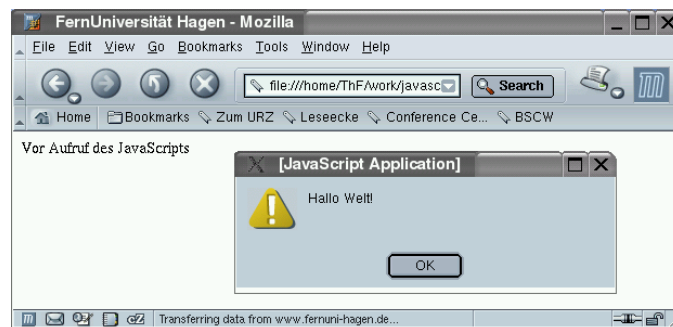


Abbildung 2.1: Ihr erstes JavaScript während der Ausführung. Was fällt Ihnen beim Betrachten des Bildes auf?

Lassen Sie das Ergebnis einen Moment auf sich wirken, bevor Sie das Popup-Fenster durch **OK** bestätigen. Wenn Sie genau aufgepasst haben werden Sie feststellen, dass die Ausgabe Ihrer HTML-Seite durch das Popup-Fenster unterbrochen worden ist – Bravo. Lassen Sie uns der Reihe nach vorgehen, was die Anweisungen in `HalloWelt.html` tatsächlich bewirken.

Zeilen 1-3 : Diese Zeilen beinhalten, was Sie über HTML bereits wissen müssten, das HTML-Dokument wird geöffnet und ein erster Absatz mit dem Inhalt `Vor Aufruf des JavaScripts` wird auf den Bildschirm gesetzt. Sollten Sie bereits an dieser Stelle Verständnisschwierigkeiten haben, sei Ihnen spätestens hier dringend die Lektüre der „[kleinen HTML-Schule](#)“^[2] nahegelegt.

² Mein geschätzter Kollege Tobias Kramer wies mich beim Korrekturlesen darauf hin, dass ich vielleicht noch erwähnen sollte, dass die im Beispiel benutzte Zeilennummerierung *nicht mitgeschrieben wird* und lediglich einer besseren Übersichtlichkeit dient.

Zeile 4 : An dieser Stelle beginnt die Einbindung des JavaScript-Codes. Alles was zwischen dieser Stelle und der `</script>`-Marke in Zeile 10 steht, wird dem JavaScript-Interpreter zur Auswertung übergeben. Der optionale Parameter `language` legt dabei fest, dass es sich bei den folgenden Anweisungen um JavaScript-Code handelt.

Gelegentlich werden Sie auch Anweisungen wie

```
<script language=javascript1.1>
```

finden. Hierdurch wird das Vorhandensein eines Interpreters einer gewissen Versionsstufe (oder neuer) zwingend gefordert. Es existiert aber nach meinem Kenntnisstand (08.09.03) kein WWW-Browser der diesen Versionszwang bis ins letzte Detail korrekt unterstützt.

Zeile 5 : Die erste JavaScript-Anweisung, ein HTML-Kommentar!

Zeilen 6, 7 und 8 : Weitere JavaScript-Anweisungen

Zeile 9 : Die letzte JavaScript-Anweisung, ein Kommentar (`/*-->`).

Zeile 10 : Ende der `<script>`-Umgebung.

Zeilen 11 und 12 : Ein weiterer HTML-Absatz und das Ende des Dokuments.

Wahrscheinlich wird Ihnen der Aufwand, der hier betrieben wird, um drei Zeilen JavaScript in das HTML-Dokument einzubetten, ziemlich hoch vorkommen. Tatsächlich ist diese Vorgehensweise jedoch ausgesprochen clever. Prinzipiell müssen wir die Welt zwischen JavaScript-verstehenden und JavaScript-ignorierenden WWW-Browsern unterteilen, und beide müssen schließlich etwas Sinnvolles aus unserem HTML-Dokument machen.

! → Browser die JavaScript *unterstützen*, stolpern in Zeile 4 über die `<script>`-Marke und wissen über den Parameter `language`, dass Sie die folgende Zeile (`<!--`) überlesen, und alles weitere bis `</script>` dem JavaScript-Interpreter überlassen können. Dieser wiederum erkennt an der Zeichenkombination `/*-->`, dass seine Aufgabe beendet ist, und er die Kontrolle an den HTML-Interpreter zurückzugeben hat. Was wir dadurch ganz nebenbei gelernt haben ist, dass die Zeichenkombination `/*` in JavaScript einen *Kommentar* bis zum Ende der jeweiligen Zeile darstellt und die Verwendung der Zeichenfolge `--` deshalb, bis auf diese Ausnahme, in JavaScript-Kommentaren verboten ist!

Browser die *nichts mit JavaScript anfangen können*, stoßen in Zeile 4 ebenfalls auf die `<script>`-Marke und tun dann das, was HTML-Interpreter mit allen Marken tun die sie nicht kennen – sie ignorieren sie einfach. Damit der darin enthaltene JavaScript-Code nun nicht als Quelltext auf dem Bildschirm erscheint, wird er nun durch das `<!--` auskommentiert; dieser Kommentar endet beim `/*-->`. Dem javascriptlosen Internet-User entgehen nun zwar unsere Kreativitätsschübe, er wird aber auch nicht durch vollkommen unverständliche Code-Anweisungen irritiert.

Wir halten also fürs Erste fest, dass die Ausführung unseres JavaScripts mit dem Auftauchen der `<script>`-Marke beginnt. Die `alert`-Anweisung erzeugt das Popup-Fenster (mit der Textkette `Hallo Welt!` als Inhalt); die weitere Ausführung des HTML-Dokuments wird solange angehalten, bis die Dialogbox mit bestätigt wird!³

`window.document.write` ist somit die zweite Ausgabeanweisung, die wir kennenlernen. Der auszugebende Text wird, inklusive aller eventuell darin enthaltenen HTML-Anweisungen, in das HTML-Dokument geschrieben.

2.3 Externes JavaScript

Anstatt den JavaScript-Code direkt in die HTML-Datei zu schreiben, können Sie ihn auch in einer externen `.js`-Datei (in unserem Beispiel `helloworldext.js`) auslagern, die dann über den `src`-Parameter mit der `<script>`-Marke verknüpft wird. Das entsprechend abgeänderte Beispiel sähe dann wie folgt aus:

```
1 // Jede Menge JavaScript-Anweisungen
2 // (gespeichert als helloworldext.js)
3 alert("Hallo Welt!");
4 window.document.write("<p>Innerhalb des JavaScripts</p>");
```

Die HTML-Datei, die nun als einbindendes Rahmendokument dient, reduziert sich wie folgt:

```
1 <html>
2 <body>
3   <p>Vor Aufruf des JavaScripts</p>
4   <script language="javascript" src="helloworldext.js">
5   </script>
6   <p>Nach Aufruf des JavaScripts</p>
7 </body>
8 </html>
```

Für welches der vorgestellten Verfahren Sie sich letztendlich entscheiden, ist Ihrem persönlichen Geschmack überlassen. Folgende Punkte sollten jedoch eventuell in Ihre Überlegungen einfließen:

- ✓ Häufig verwendeter JavaScript-Code muss nicht jedesmal neu erfasst oder kopiert, er kann direkt für mehrere HTML-Anwendungen benutzt werden.

³ Dabei scheint gerade diese Eigenschaft der `alert`-Box so nicht vorgesehen gewesen zu sein. So steht in *JavaScript, das umfassende Referenzwerk*: „... das Programm wird *nicht* angehalten; bis der Benutzer den Dialog schließt.“ Mit Ausnahme von Netscape 4.7 legen jedoch alle mir bekannten WWW-Browser beim `alert` eine Pause ein.

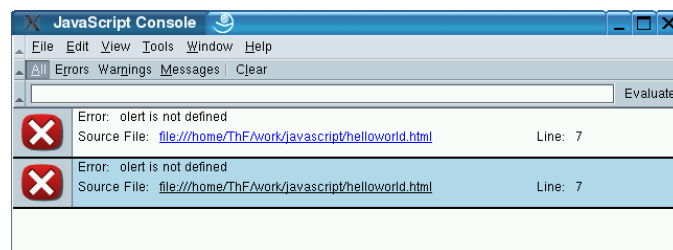


Abbildung 2.2: Debugging mit Mozilla und/oder Netscape.

- ✗ Änderungen im JavaScript-Code können dadurch aber unerwünschte Nebenwirkungen für andere HTML-Anwendungen haben.
- ✓ Der eingebundene JavaScript-Code wird in der Quelltextansicht des WWW-Browsers nicht angezeigt.

2.4 Debugging

Unter Debugging versteht man im Allgemeinen die Möglichkeit, den Ablauf eines Programms zu kontrollieren, speziell um darin enthaltene Fehler zu entdecken. Ändern Sie beispielsweise die `alert`-Anweisung in `olert`. Wie Sie sehen, sehen Sie (fast) nichts. Die beiden HTML-Anweisungen werden zwar korrekt ausgeführt, von unserem JavaScript verliert sich jedoch jede Spur.

Jetzt ist es an der Zeit, den Debugger zu starten, um den Fehler zu finden. Da bislang (leider!) kein browereinheitlicher Debug-Modus existiert, können die folgenden Beispiele nur Anhaltspunkte sein.

Mozilla und Netscape

→ Abbildung 2.2

Starten Sie den Debugger über die Menüreihenfolge **Tools**→**Web Development**→**JavaScript Console**, oder setzen Sie wahlweise den Befehl `javascript:` in der Adresszeile des Browsers ab. Sie erhalten dadurch das Debugger-Fenster mit einer (ent)sprechenden Fehlermeldung.

Wie Sie sehen, weist uns der Debugger unmissverständlich darauf hin, dass er das Kommando `olert` nicht kennt und stellt den Rest der Verarbeitung daraufhin ein. Korrigieren Sie den Fehler und laden Sie die HTML-Seite neu.

- ! → Einmal gestartet bleiben erzeugte Fehlermeldungen für die Dauer der Browsersitzung in der JavaScript Console erhalten und müssen bei Bedarf durch Betätigen der **Clear**-Taste manuell gelöscht werden. Dieses Verhalten führt häufig zu Verwirrung, wenn ein korrigiertes JavaScript-Problem den vermeintlich gleichen Fehler weiter erzeugt.

Internet Explorer

Bei Verwendung des Internet Explorers gestaltet sich der Prozess des Debuggings etwas komplexer, da Sie prinzipiell erst einmal selbst dafür ver-

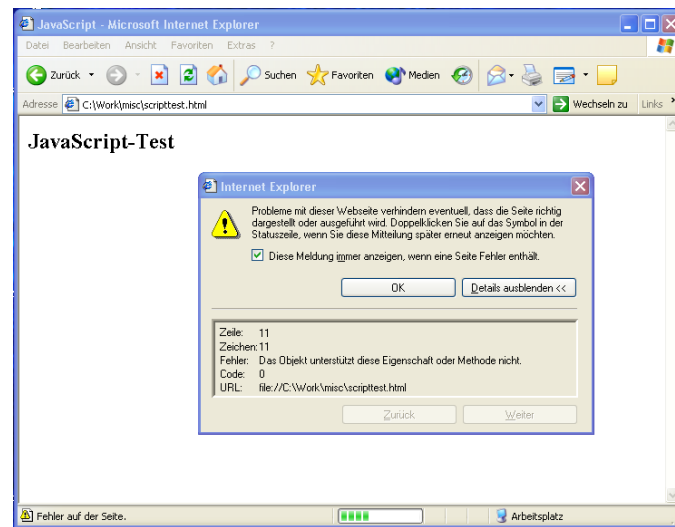


Abbildung 2.3: Debugging mit dem Internet Explorer.

antwortlich sind, dass der Debugger überhaupt aktiviert wird. Wechseln Sie daher vorab in das Menü Extras→Internetoptionen, und schauen Sie auf der Karteikarte Erweitert nach den Einstellungen für Skriptdebugging aktivieren und Skriptfehler anzeigen. Sofern es nicht bereits der Fall ist, aktivieren Sie beide Einträge.

→ Abbildung 2.3

Beim Auftreten des nächsten Skriptfehlers weist der Internet Explorer gezielt darauf hin, auch wenn die Fehlermeldungen häufig gewöhnungsbedürftig sind. Immerhin kann durch die Ausgabe der Zeilen- und Spaltennummer schnell eruiert werden, wo der Fehler aufgetreten ist.

3 Basiswissen

Man kann es drehen und wenden wie man will: Nachdem Sie im letzten Kapitel bereits erste Erfolge feiern konnten, müssen wir nun einen Gang zurück schalten und ein wenig Basisarbeit leisten – schließlich wollen Sie nicht für den Rest Ihres Lebens ausschließlich „Hello World“-Programme erstellen. Auch wenn Ihnen die folgenden Seiten etwas dröge erscheinen werden, bedenken Sie, dass auch (oder gerade?) bei einer Programmiersprache wie JavaScript die Götter den Schweiß *vor* den Erfolg gesetzt haben.

3.1 Variablen

Als Variablen bezeichnet man Speicherbereiche, in denen Sie eigene Daten ablegen, und die Sie über den vergebenen Variablennamen wieder abrufen können. Ein einmal zugewiesener Wert bleibt in einer Variable dabei so lange präsent, bis er durch einen neuen Wert überschrieben wird. Variablen werden über das Schlüsselwort `var` angelegt und können zusätzlich direkt mit einem Wert vorbelegt werden. Verdeutlichen wir dies an einem Beispiel:

```
1  <html>
2  <body>
3      <p>Vor Aufruf des JavaScripts</p>
4      <script language="javascript">
5          <!--
6
7          // Anlegen einer Variablen
8          var gruss = "Hallo Welt";
9
10         alert(gruss);
11         gruss = "Hallo Rest";
12         alert(gruss);
13         //-->
14     </script>
15     <p>Nach Aufruf des JavaScripts</p>
16 </body>
17 </html>
```

Im oberen Beispiel wird in Zeile 7 die Variable `gruss` angelegt und direkt mit der Zeichenkette `Hallo Welt` initialisiert. Nach der Ausgabe

wird die Variable mit dem Wert `Hallo Rest` neu besetzt (Zeile 9), bevor es wieder zur Ausgabe kommt. Beachten Sie, dass beim Überschreiben einer bereits vorhandenen Variable das Schlüsselwort `var` entfällt. Generell lässt sich zum Thema Variablen sagen:

Anlegen von Variablen

Neben der oben gezeigten Form, können Variablen auch ohne Initialisierung angelegt werden, beispielsweise als `var x;` Eine solche Variable besitzt bis zur Belegung den Inhalt `undefined`.

Werden mehrere Variablen benötigt (was eigentlich der Regelfall ist), so kann dies über mehrere einzelne, oder alternativ über eine Sammelanweisung geschehen. Das Ergebnis von

```
var x;  
var y;
```

und

```
var x, y;
```

ist daher identisch. Auch bei einer Sammelanweisung können Variablen initialisiert werden. Beispiel:

```
var x=1, y=4;
```

Welche dieser Schreibweisen Sie irgendwann benutzen ist mehr oder minder Geschmacksache. Aufgrund der besseren Übersichtlichkeit sollten Sie im Zweifelsfall jedoch Einzelanweisungen vorziehen.

Bezeichner

Als Bezeichner seien an dieser Stelle vorerst Variablennamen genannt; wir werden später feststellen, dass unter anderem auch Funktionen und Funktionsargumente einen Bezeichner besitzen.

Das *erste* Zeichen eines Bezeichners (und damit auch eines Variablennamens) muss immer ein Buchstabe oder der Unterstrich (`_`) sein. Die folgenden Zeichen können Buchstaben, Ziffern oder wieder der Unterstrich sein. Eine Anweisung `var 1a;` würde aus diesem Grund einen Fehler erzeugen, während `var _1_;` ein vollkommen korrekter Variablenname ist. Umlaute sind in Bezeichnern nicht erlaubt.

! → Beachten Sie auch, dass JavaScript Groß-/Kleinschreibung unterscheidet. Die Anweisung

```
var x=1, X=1;
```

erzeugt daher zwei unterschiedliche Variablen.

Datentypen

Wie beinahe alle interpreterbasierten Sprachen kennt JavaScript keine *expliziten* Datentypen, d.h. sämtlichen Variablen können beliebige Werte zugeordnet werden. Je nach Wertzuweisung oder Verwendung wird der gerade aktuelle Datentyp von JavaScript intern konvertiert um das vermeintlich günstigste Ergebnis zu erhalten. Schauen wir uns auch das einmal in einem Beispiel an:

```
1  var faktor1=3;
2  var faktor2=5;
3  var produkt;
4
5  // faktor1 und faktor2 sind Zahlen, das Ergebnis produkt
6  // wird ebenfalls als Zahl abgelegt.
7  produkt = faktor1 * faktor2;
8
9  // Ausgabe in die HTML-Datei und implizierte Konvertierung
10 // zur Textkette
11 document.write("Das Ergebnis: " + produkt);
```

Ich denke das obere Beispiel ist größtenteils selbsterklärend. Die beiden Zahlen `faktor1` und `faktor2` werden multipliziert, das Ergebnis der Variablen `produkt` zugeordnet. Verwirrung erzeugt eventuell die Ausgabeanweisung (Zeile 11), da hier vermeintlich eine *Textkette* (Das Ergebnis:) auf eine Zahl addiert wird.

Aus Sicht von JavaScript stellt sich die Vorgehensweise wie folgt dar: JavaScript erkennt die Textkette und wandelt, da Textketten und Zahlen nicht addierbar sind, die folgende Zahl implizit in eine Textkette um. Die „Addition“ bewirkt, dass beide Textketten aneinander gehängt werden.

Was aber wäre in diesem Zusammenhang passiert, wenn `faktor1` den Wert `x` anstelle von 3 gehabt hätte? Da `x` eine Zeichenkette ist und Zeichenketten und Zahlen nicht multiplizierbar sind, wäre das Ergebnis in `produkt` ein NaN für „Not a Number“ gewesen.

Der jeweilige Datentyp einer Variablen lässt sich übrigens seit JavaScript 1.1 durch die Funktion `typeof()` ermitteln, die die Werte `string`, `number`, `boolean`, `function` oder `object` zurückgibt. Welche Datentypen Sie Ihren Variablen zuweisen können, werden wir im folgenden Abschnitt sehen.

Boolsche Werte

Boolsche Variablen sind Entscheidungswerte, die nur die Zuweisungen `true` oder `false` (also *wahr* oder *falsch*) haben können.

```
var wahr=false;
var falsch=true;
```

Zahlen

Da JavaScript keine expliziten Datentypen kennt, wird auch kein Unterschied zwischen Ganzzahl- und Gleitkommazahlen gemacht. Die Festlegung wird implizit bei der Zuweisung getroffen. Folgende Zuweisungen sind möglich:

```
x = 10;           Ganze Zahl
x = 10.3;         Gleitkommazahl, Trennzeichen ist immer der Punkt
x = 10.3e-4;      Gleitkommazahl in Exponentialschreibweise
x = 0xFAB4;       dto. als Hex-Wert
```

Zeichenketten

Zeichenketten sind so ziemlich die allgemeinste Darstellungsform, da ihnen beinahe jeder Wert zugeordnet werden kann. Sie werden häufig auch als *Strings* bezeichnet. Da Zeichenketten auch Leerstellen enthalten dürfen, müssen sie bei der Zuweisung geklammert werden. Als „Quote“-Zeichen dienen dabei die einfachen (' '), bzw. die doppelten Anführungszeichen (" ").

```
var gruss="Hallo Welt";
var greeting='Hallo World';
```

Probleme bereitet die Zuweisung von Zeichen, die als *Steuerzeichen* fungieren, beispielsweise Tabulator und Zeilenschaltung, oder solchen, die JavaScript selbst benötigt, wie die oben erwähnten Anführungszeichen. Bei einer Zuweisung müssen solche Zeichen in einer Ersatzdarstellung (engl. Escape-Sequence) kodiert werden. Tabelle 3.1 zeigt eine Übersicht.

Sequenz	Englische Bezeichnung	Bedeutung
\b	Backspace	Rücktaste
\f	Form Feed	Seitenvorschub
\n	Newline	Zeilenende, Neue Zeile
\r	Carriage Return	Wagenrücklauf, Enter
\t	Tab	Tabulator
\'	Single Quote, Apostrophe	Einfaches Anführungszeichen
\"	Double Quote	Doppeltes Anführungszeichen
\\	Backslash	(Linker) Schrägstrich
\uXXXX		Beliebiges Zeichen als Unicode

Tabelle 3.1: Häufig benötigte Escape-Sequenzen

Beispiel:

```
dirs = "Verzeichnisse:\tC:\\script\n\tC:\\work";
alert(dirs);
```

würde als Ausgabe ungefähr so etwas erzeugen.

```
Verzeichnisse  C:\script
                C:\work
```

Variablen löschen

Das Löschen von Variablen wird zwar selten benötigt, es soll aber an dieser Stelle nicht unerwähnt bleiben. Seit JavaScript 1.2 existiert ein `delete`-Befehl mit dessen Hilfe nicht mehr benötigte Variablen eliminiert werden können. Beispiel:

```
delete gruss;
```

Die Variable `gruss` kann im Anschluss durch das Schlüsselwort `var` neu angelegt werden.

3.2 Arrays

Als Arrays bezeichnet man zusammenhängende Speicherbereiche gleichen Typs, auf dessen einzelne Elemente per Indizierung zugegriffen werden kann. Sofern Sie bislang noch keine Erfahrungen mit Arrays gemacht haben, will ich Ihnen zugestehen, dass Ihnen auch der obere Satz nicht unbedingt weiterhilft. Lassen Sie uns deshalb direkt in das Beispiel einsteigen.

Arrays anlegen und besetzen

Zumindest die linke Seite der folgenden Anweisung sollte Ihnen bereits vertraut sein.

```
var wochentage = new Array(7);
```

Wie Sie sehen, ist ein Array in erster Linie auch nichts anderes als eine Variable. Im Gegensatz zu dieser erfolgt momentan jedoch noch keine Wertzuweisung, sondern wir erhalten einen Speicherbereich mit sieben leeren, das heisst als `undefined` gekennzeichneten, Elementen.¹ Die nachträgliche Besetzung erfolgt manuell:

```
wochentage[0] = 'Montag';  
wochentage[1] = 'Dienstag';  
(...)  
wochentage[6] = 'Sonntag';
```

Schauen Sie sich dieses Beispiel genau an, denn es enthält gleich zwei Verfahrensweisen, die im Umgang mit Arrays ausgesprochen wichtig sind.

1. Einzelne Array-Elemente werden über den Array-Namen plus der *Position* innerhalb des Arrays angesprochen, wobei die Positionsangabe in *eckigen Klammern* steht. Die Anweisung `alert(wochentage[3])` würde also `Donnerstag` ausgeben.
- ! → 2. Das **erste Array-Element** befindet sich immer an der **Position 0**. Sofern wir also ein Array der Größe *n* anlegen, kann der Positionsanzeiger die Werte von 0 bis *n-1* besitzen!

¹ Maximal möglich wären 4.294.967.295.

Ähnlich wie Variablen können auch Arrays beim Anlegen direkt initialisiert werden.

```
var wochentage = new Array('Montag', 'Dienstag',  
    'Mittwoch', 'Donnerstag', 'Freitag',  
    'Samstag', 'Sonntag');
```

würde das gleiche Array, wie das oben in zwei Schritten angelegte, erzeugen.

Ändern der Array-Größe

Im Gegensatz zu anderen Programmiersprachen bestechen JavaScript-Arrays durch eine ausgesprochen hohe Flexibilität. So ist es beispielsweise möglich, die Größe eines Arrays noch *nach* dessen Anlegung zu ändern. Eine Schlüsselrolle spielt dabei das Attribut `length`, welches die jeweils aktuelle Array-Größe beinhaltet.

```
alert(wochentage.length);
```

würde in unserem Beispiel eine 7 ausgeben. Richtig interessant wird das Ganze jedoch, da das Attribut `length` im umgekehrten Fall auch besetzt werden darf.

```
wochentage.length = 10;
```

macht den Wunschtraum eines jeden Arbeitgebers wahr, in dem die Anzahl der Wochentage auf 10 erhöht wird. Die neuen Elemente 7, 8 und 9 werden dabei an das Ende des Arrays `wochentage` angehängt, ihr Inhalt ist `undefined`.

Umgekehrt können wir natürlich auch den Wünschen der Gewerkschaft nachkommen, in dem wir `wochentage` durch die Anweisung

```
wochentage.length = 6;
```

auf ein arbeitnehmerfreundlicheres Maß begrenzen. Wie in unserem letzten Beispiel geschieht die Größenänderung im hinteren Teil des Arrays, die Elemente 6-9 werden entfernt. Durch diese Maßnahme bleibt jedoch ausgerechnet der „Sonntag“ auf der Strecke, was der ganzen Aktion einen eher kontraproduktiven Charakter verleiht.

Mehrdimensionale Arrays

Mehrdimensionale Arrays sind in JavaScript eigentlich unbekannt. Was hier in der Theorie etwas schockierend klingt, löst sich in der Praxis jedoch relativ schnell auf. Bislang haben wir unseren Array-Elementen jeweils Basisdatentypen zugewiesen (Zeichenketten, Zahlen, ...); es spricht jedoch auch nichts dagegen einem Element einen kompletten Array zuzuweisen, was uns dann insgesamt zu dem gewünschten Ergebnis führt. Nehmen wir beispielsweise an, wir wollen unser Array `wochentage` so

umbauen, dass es jeweils die Wochentage in Deutsch und Englisch enthält, so könnten wir wie folgt vorgehen:

```
var wochentage = new Array(7);
// Montag
wochentage[0] = new Array(2);
wochentage[0][0] = "Montag";
wochentage[0][1] = "Monday";
// Dienstag
wochentage[1] = new Array(2);
wochentage[1][0] = "Dienstag";
wochentage[1][1] = "Tuesday";
...
```

Dabei bleibt festzuhalten, dass die Größe des Arrays der zweiten Dimension für jedes übergeordnete Element unterschiedlich sein darf.²

```
...
// Mittwoch
wochentage[2] = new Array(4);
wochentage[2][0] = "Mittwoch";
wochentage[2][1] = "Wednesday";
wochentage[2][2] = "Mercredi";
wochentage[2][3] = "Mercoledi";
...
```

Das Element `wochentage[2]` ist daher das einzige in unserem Array, welches *vier* Subelemente besitzt. Die Ausgabe von `wochentage[2][2]` fördert daher ein `Mercredi` zutage, während `wochentage[1][2]`, na was wohl, den Wert `undefined` besitzt. Im Fachjargon werden solche ungleichmäßig besetzten Arrays auch als *schief* bezeichnet.

3.3 Kommentare

Unter Kommentaren versteht man helfende Erläuterungen, die vom Programmierer direkt in ein JavaScript geschrieben werden. Sie sollten es sich bereits frühzeitig angewöhnen, Ihre Programme großzügig mit Kommentaren auszustatten, da dies nicht nur Dritten, die Ihre Programme lesen oder pflegen müssen, das Leben erleichtert (je nach Programmstruktur das Überleben gar erst ermöglicht); auch Sie selbst werden schnell feststellen, dass Ihre eigenen, erst vor wenigen Tagen implementierten Geniestreiche in der Zwischenzeit nicht mehr so ohne Weiteres nachvollziehbar sind.

Ein weiterer Vorteil von Kommentaren ist, dass mit ihrer Hilfe temporär nicht benötigte Programmanweisungen innerhalb des JavaScripts gewissermaßen „entschärft“ werden können, ohne dass sie gleichzeitig entfernt werden müssen – sie werden einfach *auskommentiert*.

² Präzise formuliert müssen es noch nicht einmal durchgehend weitere Arrays sein.

Selbstverständlich muss dem JavaScript-Interpreter klar gemacht werden, dass er das kommentierende Beiwerk zu ignorieren hat. Dies geschieht durch vorangestellte, spezielle Zeichenkombinationen. JavaScript kennt gleich zwei Möglichkeiten, Kommentare zu hinterlegen.

Einzeiliger Kommentar (//): Die vorangestellten Slashes habe ich in dieser Broschüre bereits häufiger benutzt. Der JavaScript-Interpreter erkennt an ihnen, dass der folgende Text bis zum *Ende der Zeile* als Kommentar betrachtet werden soll.

Blockkommentar (/*...*/): Erkennt der JavaScript-Interpreter das Kommentarzeichen „/*“, so wird der folgende Text solange überlesen, bis die Endesequenz „*/“ erscheint. Der Blockkommentar eignet sich daher ausgezeichnet um längere Programmpassagen auszukommentieren.

```

1  /*****
2  * comment.js - dient eigentlich nur als ausführliches
3  * Beispiel für die Kommentierung
4  *
5  * Autor: ThF, 8/03
6  *****/
7
8  var ausgabe = "Kommentare sind sinnvoll!";
9  // Die folgende Anweisung wird auskommentiert, da auch
10 // eine direkte Ausgabe möglich ist
11 /* document.write(ausgabe); */
12 document.write("Kommentare sind sinnvoll");

```

3.4 Zuweisungen

Obwohl wir in den oberen Beispielen schon laufend Zuweisungen benutzt haben, sollten wir sie an dieser Stelle noch einmal ausführlicher betrachten. Generell erfolgt die Zuweisung durch das Gleichheitszeichen (=), wobei dem *links* davon stehenden Ausdruck der Wert des auf der *rechten* Seite befindlichen Ausdrucks zugewiesen wird. Der hierbei verwendete Begriff „Ausdruck“ deutet dabei schon an, dass es sich links und rechts vom Gleichheitszeichen nicht immer unbedingt um Variablen und zuzuweisende Konstanten handeln muss; eine Zuweisung wird aber in jedem Fall durch ein *Semikolon* (;) beendet. Schauen wir uns daher einmal die beiden folgenden Zuweisungen an.

```

var y = 7;
var a = 3;
/* Zuweisungen */
x = 5 * y;
Z = a = x;

```


Die ersten beiden An- bzw. Zuweisungen sollten Ihnen mittlerweile keine Bauchschmerzen mehr bereiten. Hier werden jeweils den Variablen `y` und `a` feste Zahlenwerte zugewiesen.

In Zeile 4 wird der Wert von `y` vor der Zuweisung mit 5 multipliziert. Das Ergebnis (35) wird in der Variablen `x` abgelegt.

Zeile 5 macht uns mit dem Begriff der *Reihenzuweisung* bekannt. Halten Sie sich bei solchen Konstrukten vor Augen, dass stets von *rechts nach links* ausgewertet und zugewiesen wird, d.h. vorab erhält `a` den Wert von `x` (35) und erst nach dieser Zuweisung `z` den Wert von `a`.

! → Für den Fall, dass die Variable `y` in Zeile 5 einen Wert besitzt der eine Multiplikation unmöglich macht (beispielsweise ein `Hallo`), so erzeugt die Zuweisung den Wert `NaN` für „Not a Number“. Dieser Wert kann über die Funktion `isNaN()` abgefragt werden. Der Ausdruck

```
x = isNaN(5 * 7);
```

würde der Variable `x` den boolschen Wert `false` zuweisen!

Generell macht es für den JavaScript-Interpreter keinen Unterschied, ob Sie für jede Zuweisung eine einzelne Zeile reservieren oder Ihr gesamtes JavaScript hintereinander weg schreiben. Für den Interpreter ist das Semikolon und nicht das Zeilenende entscheidend.

```
var y = 7; var a = 3; /* Zuweisungen */ x = 5 *  
y; z = a = x;
```

Sie werden mir beipflichten, dass Programmcode der oben aufgeführten Art nicht unbedingt die Lesbarkeit erhöht.

3.5 Arithmetische Operationen

Neben der im letzten Kapitel bereits vorgestellten *Multiplikation* (`x*y`), können Sie in JavaScript auch die restlichen drei Grundrechenarten (`x/y`, `x+y`, `x-y`) benutzen. Zusätzlich gibt es den *Modulo*-Operator (`x%y`), der den *ganzzahligen*, nicht mehr durch den Divisor dividierbaren, *Rest* zurück gibt.³

In Verbindung mit, oder als Ersatz für Zuweisungen gibt es jedoch auch eine Reihe von *verkürzten Schreibweisen*:

```
x++, ++x    für x = x + 1  
x--, --x    für x = x - 1  
x += y      für x = x + y  
x *= y      für x = x * y  
...
```

! → Beachten Sie unbedingt, dass die Varianten für die Erhöhung, bzw. Erniedrigung um 1, in einer Verbindung mit einer weiteren Zuweisung unterschiedliche Ergebnisse liefern, da in der ersten Form (`y=x++`) erst der

³ Für alle diejenigen, deren Mathewissen längere Zeit brach gelegen hat: $11\%8=3$.

Wert von `x` zugewiesen wird und danach die Erhöhung statt findet, während bei `y=++x` zuerst der Wert von `x` geändert wird, *bevor* im Anschluss die Zuweisung erfolgt. Sie sollten im Sinne eines sauberen Programmierstils Konstrukte, wie die zuletzt gezeigten, nach Möglichkeit meiden.

3.6 Mathematische Funktionen

Der Vollständigkeit halber, und weil es thematisch gerade ganz gut passt, seien an dieser Stelle noch ein paar mathematische Funktionen erwähnt, die über die oben aufgeführten Grundrechenarten hinausgehen.

<code>x = Math.random()</code>	erzeugt eine Zufallszahl x , mit $0 \leq x < 1$
<code>Math.abs(x)</code>	Betrag von x
<code>Math.round(x)</code>	x auf die nächste Ganzzahl gerundet
<code>Math.ceil(x)</code>	x auf die nächste <i>größere</i> Ganzzahl gerundet
<code>Math.floor(x)</code>	x auf die nächste <i>kleinere</i> Ganzzahl gerundet
<code>Math.min(x, y)</code>	Minimum von x und y
<code>Math.max(x, y)</code>	Maximum von x und y
<code>Math.sqrt(x)</code>	\sqrt{x}

Weiterhin existieren unter anderem noch Funktionen für Sinus, Cosinus, Logarithmus, sowie Konstanten für die Eulersche Zahl e und die Ludolf-sche Zahl π .

3.7 Funktionen für Zeichenketten

Zeichenketten (oder Strings) genießen in JavaScript einen gewissen Sonderstatus da sie, wie auch in anderen Programmiersprachen üblich (beispielsweise Java oder C), intern eigentlich als ein Array aus Charactern dargestellt werden. Die daraus üblicherweise resultierenden Probleme, wie Besonderheiten beim Vergleich, bleiben Ihnen bei JavaScript erspart, so dass Sie Zeichenketten im Großen und Ganzen wie „normale Datentypen“ behandeln können. Trotzdem existiert eine Reihe von Funktionen, die Sie bei der Verarbeitung von Zeichenketten unterstützen.

Ermittlung der Zeichenanzahl

An dieser Stelle stoßen Sie erstmalig auf etwas Vertrautes. Da, wie oben bereits angesprochen, eine Zeichenkette eigentlich ein Array von Einzelzeichen ist, ist es wohl keine Überraschung, dass zur Ermittlung der Länge eines Strings das bereits bekannte Attribut `length` benutzt wird.

```
var s = "Hallo Welt";
var l = s.length; // ergibt 10
l = "Hallo Welt".length // ebenso
```

Ein Zeichen ermitteln

Syntax: *Zeichenkette*.charAt(*index*)

Mit Hilfe der Funktion `charAt` können einzelne Zeichen in einer Zeichenkette ermittelt werden. Dabei bezeichnet *index* die Position des Zeichens innerhalb des Strings. Analog zu Arrays besitzt die erste Position der Zeichenkette den Wert 0, das letzte Zeichen befindet sich an der Stelle *Zeichenkette*.length-1.

```
var s = "Hallo Welt";  
var l = s.charAt(7); // ergibt e
```

Für $index < 0$ oder $index \geq \text{Zeichenkette.length}$ ist das Ergebnis der Operation die *leere* Zeichenkette (`""`).

Zeichenketten verknüpfen

Wie bereits gesehen können Zeichenketten mit Hilfe des `+`-Operators aneinander gehängt werden. Als Ergebnis entsteht eine neue Zeichenkette, sobald einer der beteiligten Operanden eine Zeichenkette ist.

Alternativ kann jedoch auch die Methode `concat` verwendet werden.

```
var s = "Pi ist ungefähr:";  
var x = 3.14;  
var e1 = s + " " + x; // Pi ist ungefähr: 3.14  
var e2 = s.concat(" ").concat(x); // dto.
```

Teilzeichenketten bilden

Syntax: *Zeichenkette*.substring(*start*, *ende*)
Zeichenkette.substr(*start*, *länge*)

`substring` und `substr` sehen sich auf den ersten Blick sehr ähnlich, reagieren in der Anwendung aber vollkommen unterschiedlich.

substring: beginnt mit der Erstellung der Teilzeichenkette bei Position *start*. Das Zeichen an der Position *ende* ist nicht mehr Teil des neu gebildeten Strings! Für den Fall $start > ende$ werden die Parameter getauscht.

substr: beginnt ebenfalls an der Position *start*, und schneidet die folgenden *länge* Zeichen aus.

```
var s = "Hallo Welt!!";  
var sub1 = s.substring(1,7); // ergibt "allo W"  
var sub2 = s.substr(1,7); // ergibt "allo We"  
sub1 = s.substring(7,1); // ergibt "allo W"  
sub2 = s.substr(7,1); // ergibt "e"
```

In beiden Fällen wird durch Weglassen des zweiten Parameters bis zum Ende der ursprünglichen Zeichenkette gelesen.

Zeichenketten aufteilen

Syntax: *Zeichenkette*.split(*trennzeichen*)

Diese Methode `split` erzeugt ein Array von Teilzeichenketten, in dem sie jeweils ein neues Element erzeugt, sobald in *Zeichenkette* die Teilzeichenkette *trennzeichen* gefunden wird.

```
var s = "1,2,3,4,5";
var zar = s.split(",");
document.write(zar[0]); // ergibt 1
document.write(zar[1]); // ergibt 2
...
```

Teilzeichenketten suchen

Syntax: *Zeichenkette*.indexOf(*suchstring*, *start*)

`indexOf` sucht in *Zeichenkette* ab Position *start* nach dem Auftreten von *suchstring*. Zurückgegeben wird die Position des ersten Zeichens von *suchstring* in *Zeichenkette*, oder -1 sofern *suchstring* nicht in *Zeichenkette* enthalten ist.

Wird das zweite Argument (*start*) weggelassen, beginnt die Suche an der Position 0.

```
var s = "Hallo Welt";
var tk = s.indexOf("llo W"); // ergibt 2
tk = s.indexOf("llo X"); // ergibt -1
```

Zeichenketten mit HTML-Formatierungen

Eine der Hauptaufgaben von JavaScript besteht, wie wir bereits öfter gesehen haben und noch ausführlicher sehen werden, darin, dynamischen Text in HTML-Ausgaben zu schreiben. Dies stellt im Normalfall kein Problem dar und kann prinzipiell über die Verkettung von Zeichenketten realisiert werden.

```
var s = "Hallo Welt";
document.write("<i>" + s + "</i>");
// ...schreibt den Text in italic
```

Da diese Art der formatierten Ausgabe einigermaßen umständlich erscheint, werden wir von JavaScript mit einigen passenden Methoden unterstützt. Das oben aufgeführte Beispiel ließe sich wie folgt verkürzen:

```
var s = "Hallo Welt";
document.write(s.italics());
// ...schreibt den Text in italic
```

Die folgenden Methoden eignen sich zur Umsetzung einer Zeichenkette *Zk* in direkte HTML-Formate

JavaScript	HTML
<code>Zk.big()</code>	<code><BIG>Zk</BIG></code>
<code>Zk.bold()</code>	<code>Zk</code>
<code>Zk.fixed()</code>	<code><TT>Zk</TT></code>
<code>Zk.fontcolor(Farbe)</code>	<code>Zk</code>
<code>Zk.fontSize(Größe)</code>	<code>Zk</code>
<code>Zk.italics()</code>	<code><I>Zk</I></code>
<code>Zk.link(URL)</code>	<code>Zk</code>
<code>Zk.small()</code>	<code><SMALL>Zk</SMALL></code>

Zeichenketten und Zahlen

Wie bereits erwähnt, kann JavaScripts interne Datentypkonvertierung gelegentlich zu eher unerwünschten Ergebnissen führen. Betrachten wir folgendes als Beispiel:

```
var x = 1;
var y = 7;

document.write("Die Summe ist: " + x + y);
```

Als Ausgabe der `write`-Anweisung erfolgt nicht die erwartete 8, sondern eine 17, da JavaScript in einer durch das `write` eingeleiteten Zeichenkettenoperation beide Variablen als Zeichenketten betrachtet. Aus diesem Grund seien hier noch drei Funktionen vorgestellt, die bei Bedarf aus Zeichenketten wieder Zahlen machen.

- `eval(Ausdruck)` versucht *Ausdruck* als mathematische Anweisung zu interpretieren. Im oberen Beispiel hätte die Anweisung `eval(x + y)` für das richtige Ergebnis gesorgt.
- `parseInt(zk)` versucht aus der Zeichenkette *zk* eine ganze Zahl zu machen. Ist dies nicht möglich wird als Rückgabewert `NaN` erzeugt.
- `parseFloat(zk)` versucht aus der Zeichenkette *zk* eine Gleitkommazahl zu machen. Ist dies nicht möglich wird als Rückgabewert `NaN` erzeugt.

! → `parseInt(3.14)` erzeugt als Ergebnis eine 3!

Selbstverständlich ist auch der umgekehrte Fall denkbar, nämlich dass Zahlen bewusst als Zeichenketten abgespeichert werden sollen, bzw. Zahlen in Zeichenketten konvertiert werden sollen. Hier hilft das explizite Anlegen eines Objekts vom Typ `String()`.

Beispiel um den numerischen Inhalt der oben angelegten Variable `x` in eine Zeichenkette zu überführen:

```
var zkx = new String(x);
```



Abbildung 3.1: prompt, Die einzige Möglichkeit Text in ein laufendes JavaScript zu bringen.

3.8 Einfache Ein- und Ausgaben

Wir wollen dieses Kapitel mit der Betrachtung von Ein-/Ausgaben unter JavaScript beschließen. Prinzipiell lassen sich in diesem Komplex die folgenden Möglichkeiten unterscheiden:

3.8.1 Eingaben

Da JavaScript-Programme, rein technisch betrachtet, Skripte sind, die von fremden Servern geladen, und auf Ihrem Rechner ausgeführt werden, sind die Möglichkeiten der Interaktion mit Ihrem Rechner auf ein Minimum reduziert. Da der JavaScript-Interpreter aus Sicherheitsgründen keine lesenden Zugriffe auf Ihr lokales Dateisystem erlaubt, bleibt als einzige Möglichkeit eine einzeilige Texteingabe über ein Eingabefenster. Dieses kann aus einem JavaScript heraus mit der Methode `window.prompt` erzeugt werden.

Syntax: `window.prompt(Meldungstext, Vorgabe)`

Dabei erscheint *Meldungstext* als Beschriftung für ein einzeiliges Textfeld, welches beim Erscheinen bereits den Wert *Vorgabe* haben kann. Soll keine Voreinstellung angezeigt werden, so ist als zweiter Parameter eine leere Zeichenkette ("") anzugeben.

→ Abbildung 3.1

Zusätzlich erhält das Meldungsfenster die Schalter **OK** (liefert bei Betätigung den im Eingabefeld befindlichen Text) und **Abbrechen** (erzeugt ein null-Objekt).

```
var antwort =  
window.prompt("Sind Sie JavaScript-Fan?", "Ja");
```

Auf Layout und Aufbau des `prompt`-Fensters haben Sie keinen Einfluss, sie werden vom Browser bestimmt.

3.8.2 Ausgaben

Im Bereich von Textausgaben sind Sie nicht viel, aber doch ein wenig flexibler als dies bei der Eingabe der Fall ist. Generell besitzen Sie gleich drei Möglichkeiten eventuell entstehende Textausgaben unterzubringen

Ausgabe auf die Web-Seite

Die Möglichkeit Zeichenketten direkt im HTML-Code unterzubringen, haben wir in diesem Kapitel schon häufig strapaziert. Eine Schlüsselrolle spielt dabei die Methode `write`, die bislang abwechselnd als `document.write` oder `window.document.write` benutzt wurde.

Die komplette Syntax lautet

```
window.document.write("Text ")
```

wodurch die Zeichenkette *Text*, die wie gesehen auch HTML-Sequenzen enthalten kann (und soll), im laufenden Ausgabestrom untergebracht wird. Das dies letztendlich durch das `write` geschieht ist wohl einleuchtend, welche Aufgaben aber haben dann `document` und `window`?

Um die Lösung etwas abzukürzen: Beide geben an, wohin *Text* de facto geschrieben wird. `window` ist dabei eine Referenz auf das Browserfenster in dem das JavaScript gestartet worden ist – momentan also Ihr kompletter WWW-Browser. Wir werden später Möglichkeiten kennenlernen, zusätzliche Fenster zu öffnen und auch zu beschreiben. Da wir es bislang ausschließlich mit *einem* Fenster zu tun haben, könnten wir die explizite Angabe von `window` momentan jedoch auch weglassen.

`document` bezeichnet den Teil von `window`, in den die Ausgabe geschrieben wird. Dies macht in dem Augenblick schlagartig Sinn, in dem uns klar wird, dass unser `window` nicht nur aus dem Bereich für die WWW-Seite besteht, sondern dass es darüberhinaus auch noch eine Titelleiste, eine Navigationsleiste und vieles Andere mehr gibt.

Fassen wir für den Moment zusammen: Um die Ausgabe einer Überschrift der ersten Ordnung in unserer HTML-Seite zu erzeugen, können wir die Anweisung

```
window.document.write("<H1>Hallo</H1>");  
// oder alternativ:  
document.write("<H1>Hallo</H1>");
```

benutzen.

Ausgabe in die Statuszeile

→ Abbildung 3.2 auf der nächsten Seite

Als weiteres beschreibbares Element von `window` dient neben dem Dokument-Bereich auch die *Statuszeile* – im Normalfall am unteren Fensterrand befindlich. Da die Statuszeile in aller Regel eher begrenzt aufnahmefähig ist, existiert zum Beschreiben keine Methode, es reicht wenn die `window`-Eigenschaft `status` besetzt wird.

```
window.status = "Diese Seite benutzt JavaScript";
```

wo er verbleibt, bis er durch eine neue `status`-Meldung ersetzt wird. Daneben sorgt im Normalfall auch ein *Fokuswechsel* für einen Status.



Abbildung 3.2: Ein JavaScript kann Meldungen in die Statuszeile des Browsers schreiben, hier am Beispiel von Mozilla.

Da wir uns bislang aber noch nicht mit Fokussierung auseinander gesetzt haben, reicht an dieser Stelle der Hinweis, dass in diesem Fall der von uns gesetzte Status durch einen Default-Wert abgelöst wird. Dieser Default-Wert ist normalerweise die leere Zeichenkette (""), was kann aber benutzerseitig durch die Besetzung von `window.defaultStatus` geändert werden kann.

Meldungsfenster

→ Abbildung 2.1 auf Seite 8

Das JavaScript zur Ausgabe von Meldungen relativ unkompliziert eigene Fenster erzeugen kann, haben wir bereits zu Beginn dieser Broschüre gesehen. Die simpelste Form stellt dabei die `alert`-Anweisung dar.

```
window.alert("Hallo Welt");
```

→ Abbildung 3.3

Etwas ganz Ähnliches wird durch die Methode `confirm` dargestellt, die ein Meldungsfenster mit zwei Bestätigungsknöpfen aufbaut.

```
var antwort = window.confirm("Diese Seite benutzt  
Javascript?");
```

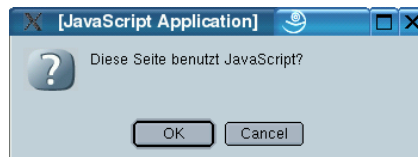


Abbildung 3.3: Ein Bestätigungsfenster mit zwei Schaltern, der betätigte Knopf kann abgefragt werden.

`confirm` liefert, im Gegensatz zu `alert`, einen booleschen Wert zurück dem entnommen werden kann ob auf die Taste **OK** (`true`), oder **Abbruch** (`false`) gedrückt wurde.

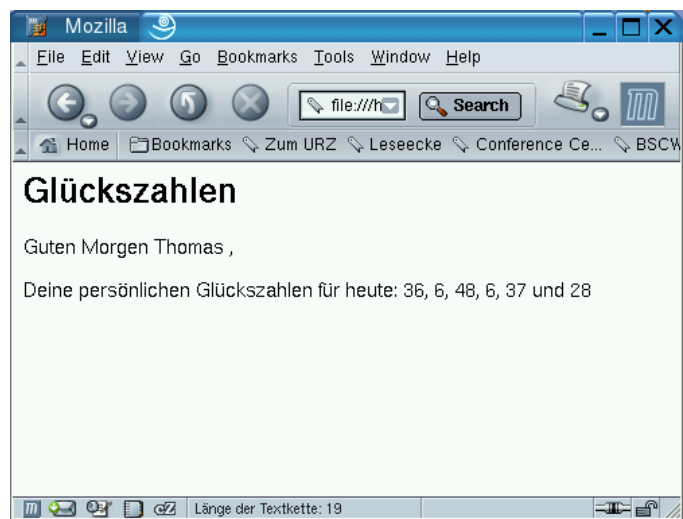
3.9 Übungsaufgaben

1. Erstellen Sie ein JavaScript, das über ein `prompt`-Fenster einen Vornamen einliest. Der eingebene Vorname soll im Anschluß mit dem Gruß Guten Morgen auf der Web-Seite erscheinen.

Berechnen Sie zusätzlich die Länge der gesamten Textkette, und geben Sie diese in der Statuszeile des Browserfensters aus.

2. Geben Sie eine zusätzlich Meldung auf der Web-Seite mit dem Inhalt `Deine persönlichen Glückszahlen für heute: und sechs` zufällig ermittelten Zahlen im Bereich zwischen 1 und 49 aus.

Das komplette Ergebnis sehen Sie im unten stehenden Bild.



4 Kontrollstrukturen

An dieser Stelle haben Sie schon schon eine ganze Menge erreicht. Sofern Sie das nicht glauben wollen: Sie können bereits jetzt JavaScript-Prozeduren in HTML-Dokumente einbetten und ausführen, Sie können innerhalb dieser JavaScripte Variablen und Arrays besetzen, Sie können Variableninhalte verändern, und nicht vergessen sollten wir, dass Sie Ihre JavaScripte durch die Hilfe von Einleseprozeduren mit dynamischen Werten versehen und diese auch ausgeben können. Wie gesagt, Sie haben schon eine Menge erreicht.

Wir wollen uns im Weiteren mit der Aufgabe beschäftigen, welche Einflussmöglichkeiten Sie auf den Ablauf Ihres JavaScripts haben. Sofern Sie keine Vorstellung davon haben, was ein *Programmablauf* ist, versetzen Sie sich am besten geistig zum Domino-Day von RTL. Bei dieser Veranstaltung geht es im Großen und Ganzen darum, dass nach Umwerfen eines Startsteins eine Kettenreaktion ausgelöst wird, an deren Ende ein jubelnder Pulk von Schülern und Studenten steht, die es gerade geschafft haben eine Flugzeughalle mit 42.000.000.000.000.000.000.000.000 umgeworfenen Dominosteinen zu füllen.



Genau wie bei dieser Dominosteinkette wird ein JavaScript an einer Stelle eröffnet (Programmstart) und endet an einem festgelegten Punkt (Programmende), zwischen Programmstart und -ende befindet sich der Programmablauf. Während des Ablaufs können die Dominosteine *Verzweigungen* einschlagen (Fallunterscheidung), in Kreisen laufen (Schleifen) und sogar mehrere Ablauflinien eröffnen (Multithreading). Die Möglichkeit eines vorzeitigen Abbruchs der Dominokette (Programm Error) ist ebenfalls vorstellbar, beispielsweise wenn zwei Steine zu weit auseinander stehen. Nicht unerwähnt bleiben sollte auch die Freude des Programmierers nach einem fehlerfreien Durchlauf, bzw. die entsprechende Frustration im umgekehrten Fall.

Wir verlassen daher nach dieser Erklärung die rauchende Flugzeughalle, und beschäftigen uns mit den Möglichkeiten, den Programmablauf mit JavaScript zu kontrollieren.

4.1 Verzweigungen und Fallunterscheidungen

If ... then ... else

Die klassischste aller Fallunterscheidungen ist wohl das *WennDann*-Konstrukt. Hierbei wird jeweils eine Bedingung abgefragt, deren Ergebnis `true` oder `false` sein kann – in beiden Fällen kann jeweils unterschiedlicher Programmcode ausgeführt werden. Beispiel:

```
var grenzwert = 40;
var alter = window.prompt("Ihr Alter?", "");

if (alter < grenzwert) {
    alert("Sie sind noch beinahe jugendlich.");
}
```

Im oberen Beispiel wird die Fallunterscheidung durch das Schlüsselwort `if` eingeleitet, worauf die *Bedingung* in Klammern folgt. Für den Fall, dass die Bedingung erfüllt ist, also ein `true` zurückgibt, werden die Anweisungen in den *geschweiften Klammern* (`{}`) ausgeführt; ansonsten wird dieser Bereich übersprungen. Zusätzlich kann auf den Anweisungsteil eine `else`-Anweisung folgen, deren Ausführung beginnt, sofern das Ergebnis der Bedingung `false` ist.

```
if (alter < grenzwert) {
    alert("Sie sind noch beinahe jugendlich.");
} else {
    alert("Sie sollten sich Gedanken um Ihre Altersversorgung machen!");
}
```

Als Vergleichsoperatoren können Sie neben dem oben gezeigten *kleiner* (`<`) noch die Folgenden verwenden:

<code>></code>	prüft auf <i>größer als</i>
<code>>=</code>	prüft auf <i>größer als oder gleich</i>
<code><=</code>	prüft auf <i>kleiner als oder gleich</i>
<code>==</code>	prüft auf <i>identisch</i>
<code>!=</code>	prüft auf <i>verschieden</i>
<code>!(Bedingung)</code>	<i>negiert</i> den Wert von <i>Bedingung</i>

! → Beachten Sie unbedingt, dass der Identitätsvergleich durch *zwei Gleichheitszeichen* (`==`) dargestellt wird. Die Verwendung *nur eines* Gleichheitszeichens ist eine *Zuweisung*. JavaScript erzeugt bei einem Konstrukt wie

```
if (alter = grenzwert) {...
```

keine Fehlermeldung, sondern weist stattdessen der Variable `alter` den Inhalt von `Grenzwert` zu. Das Ergebnis dieser Zuweisung ist in jedem Fall `true`!

Weiterhin können *mehrere Bedingungen* miteinander verknüpft werden. Hierzu werden die *logischen Operatoren* „&&“ als **und**, bzw. „||“ für **oder** benutzt. Beispiel:

```
if ( (alter > 25) && (alter < 45) ) {  
    alert("Sie sind im besten Alter.");  
}
```

! → Beachten Sie jedoch:

und(&&): Die Gesamtbedingung ist *wahr*, wenn *alle* Teilbedingungen *true* zurückgeben. Ist bereits die erste Teilbedingung *falsch*, werden die restlichen Teilbedingungen nicht mehr ausgeführt.

oder(||): Die Gesamtbedingung ist *wahr*, wenn *eine* Teilbedingung *true* zurückgibt. Ist bereits die erste Teilbedingung *wahr*, werden die restlichen Teilbedingungen nicht mehr ausgeführt.

Die oben gezeigten Vergleiche gehören zu den beliebtesten Möglichkeiten um das Ergebnis einer Bedingung zu erzeugen. Es sind aber prinzipiell, und wie bereits oben gesehen, alle Zuweisungen möglich, die einen Booleschen Wert erzeugen. So würde auch folgendes Konstrukt funktionieren:

```
if (window.confirm("Verraten Sie uns Ihr Alter?")) {  
    alert("Wunderbar, Sie haben nichts zu verbergen.");  
} else {  
    alert("Dann eben nicht.");  
}
```

Selektion

Eine zweite Form der Fallunterscheidung besteht in der Selektion, in der eine Variable gleich mit mehreren Werten verglichen werden kann. Die allgemeine Form besitzt folgendes Aussehen:

```
switch(Ausdruck) {  
    case Wert1:  
        /* Anweisungen für Ausdruck == Wert1 */  
    case Wert2:  
        /* Anweisungen für Ausdruck == Wert2 */  
    (...)   
    default:  
        /* Anweisungen für alle anderen Werte */  
}
```

Vom Prinzip her scheint diese Struktur erst einmal verständlich. Ist ein *links* vom Doppelpunkt (:) stehender Wert mit dem abgefragten Ausdruck identisch, so werden die *rechts* vom Doppelpunkt befindlichen Anweisungen ausgeführt. Wird *ausdruck* durch keinen der angegebenen Werte erfüllt, so greifen als letzte Instanz die Anweisungen für den Wert *default*.

- ! → Eine Eigenart der `switch`-Anweisung, die häufig für Verwirrung sorgt, kommt uns an dieser Stelle noch in die Quere. Ist die Ausdruck/Wert-Beziehung ersteinmal erfüllt, d.h. es werden Anweisungen auf der *rechten* Seite ausgeführt, so werden ab diesem Punkt *alle* dort stehenden Anweisungen abgearbeitet, und zwar unabhängig von den noch links stehenden Werten – die `switch`-Anweisung *fällt durch*!

Betrachten wir zur Verdeutlichung das folgende Beispiel:

```
var meld = "";
switch(window.prompt("Wie spät ist es?", "")) {
    ...
    case 8:    meld = "Es ist acht Uhr";
    case 9:    meld = "Es ist neun Uhr";
    case 10:   meld = "Es ist zehn Uhr";
    ...
}
document.write(meld);
```

In diesem Fall würde die Variable `meld`, unabhängig von der eingegebenen Uhrzeit, immer die zuletzt zugewiesene Uhrzeit-Meldung beinhalten, in unserem Beispiel also immer `Es ist zehn Uhr`. Um diesen unerwünschten Effekt zu umgehen, müssen auf der rechten Seite `break`-Anweisungen gesetzt werden, welche die weitere Verarbeitung unterbrechen.

```
var meld = "";
switch(window.prompt("Wie spät ist es?", "")) {
    ...
    case 8:    meld = "Es ist acht Uhr";
               break;
    case 9:    meld = "Es ist neun Uhr";
               break;
    case 10:   meld = "Es ist zehn Uhr";
               break;
    ...
}
document.write(meld);
```

Daneben kann es natürlich auch Situationen geben, in denen das Durchfallen der `switch`-Anweisung geradezu erwünscht ist, wie das unten stehende, modifizierte Beispiel zeigt.

```
var meld = "Es ist ";
switch(window.prompt("Wie spät ist es?", "")) {
// falls through...
    case 8:
    case 9:
    case 10:
    case 11:   meld = meld + "Morgen";
               break;
```

```

        case 12: meld = meld + "Mittag";
                break;
        case 13:
        case 14:
        ...
        case 17: meld = meld + "Nachmittag";
                break;
        ...
    }
    document.write(meld);

```

Hier würde, bei Eingabe der Zahlen 8, 9, 10 oder 11 immer die Meldung *Es ist Morgen* auf dem Bildschirm erscheinen, jedoch nicht ein zusätzliches *Es ist Mittag*, da die um 11 Uhr eingestreute *break*-Anweisung ein weiteres Durchfallen verhindert.

Es zeugt ganz nebenbei von einem guten Programmierstil, durchfallende *switch*-Anweisungen mit einer Kommentarzeile „falls through...“ zu versehen.

4.2 Schleifen

while-Schleifen

Mit Hilfe von Schleifen kann der Programmablauf in eine Richtung gesteuert werden, dass ein Teil der darin enthaltenen Anweisungen solange wiederholt wird, bis eine bestimmte Bedingung erfüllt ist. Der Prototyp einer Schleife wird dabei durch die klassische *while*-Anweisung repräsentiert. Im folgenden Beispiel können wir nun eine simple Plausibilitätsprüfung realisieren:

```

var ein = prompt("Bitte eine Zahl eingeben:", "");
var zahl = parseInt(ein);

while(isNaN(zahl)) {
    ein = prompt("Bitte eine Zahl eingeben:", "");
    zahl = parseInt(ein);
}

/* Weiterer Programmablauf */

```

Durch die *while*-Schleife wird die weitere Ausführung des Programmablaufs solange verhindert, bis der Anwender eine Zahl eingegeben hat – ein Verfahren, welches in der Praxis häufiger benötigt wird als man vermuten würde. Im nächsten Beispiel nutzen wir unsere *while*-Schleife um die Quadratzahlen von 1 bis 20 zu berechnen.

```

var i = 1;

while(i <= 20) {

```

```

        document.write("zahl: " + i +
        "Quadratzahl: " + i*i + "<BR>");
        i++;
    }

```

! → Beachten Sie unbedingt, dass die Abbruchbedingung einer *while*-Schleife irgendwann erfüllt werden muss, da Sie ansonsten eine *Endlosschleife* produzieren!¹

for-Schleifen

Das zuletzt gezeigte Beispiel lässt sich durch die Verwendung einer *for*-Schleife etwas eleganter gestalten.

```

var i;

for(i=1; i <= 20; i++) {

    document.write("zahl: " + i +
    "Quadratzahl: " + i*i + "<BR>");

}

```

Der Vorteil der *for*-Schleife besteht in diesem Beispiel darin, dass sich sämtliche Kriterien zur Schleifensteuerung nun im Kopf der Schleife befinden, während bei Verwendung der *while*-Schleife die Erhöhung des Zählers im Schleifenrumpf stattfindet. Prinzipiell besteht der Schleifenkopf einer *for*-Schleife aus *drei* Anweisungsteilen.

```

for (Initialisierung;
    Abbruchbedingung;
    Durchgangsende) {

```

Die einzelnen Anweisungsteile werden dabei durch ein *Semikolon* (;) voneinander getrennt. Sollen/müssen in einem Anweisungsteil mehrere Anweisungen ausgeführt werden, so werden diese durch ein *Komma* (,) voneinander getrennt. Eine solche Vorgehensweise erhöht jedoch nicht unbedingt die Lesbarkeit des Konstrukts, wie das abschließende Beispiel zeigt.

```

for(var i=1; i <= 20;
    document.write("zahl: " + i +
    "Quadratzahl: " + i*i + "<BR>"),
    i++) {
    /* Schleifenkörper bleibt leer */
}

```

¹ Sie können die oben aufgeführte Schleife testweise ausprobieren, indem Sie die Anweisung *i++* auskommentieren.

Weitere Schleifen

Eine Variante zur oben gezeigten `while`-Schleife ist die `do`-Schleife mit Fußprüfung. Im Gegensatz zur `while`-Schleife wird die `do`-Schleife *mindestens einmal* durchlaufen.

```
var ein;  
var zahl;  
  
do {  
  
    ein = prompt("Bitte eine Zahl eingeben:", "");  
    zahl = parseInt(ein);  
  
} while(isNaN(zahl));
```

- ! → Die `do`-Schleife mag häufig, und speziell für dieses Beispiel, einfacher als die `while`-Schleife aussehen. Im praktischen Gebrauch ist sie jedoch in aller Regel unübersichtlicher als ihr Äquivalent mit der Kopfprüfung. Sie sollte daher im Normalfall nicht verwendet werden, insbesondere weil sich *jede* `do`-Schleife auch als `while`-Schleife darstellen lässt.

4.3 Übungsaufgaben

1. Erstellen Sie ein Programm das alle vierstelligen Zahlen $abcd$ findet für die gilt:

$$ab^2 + cd^2 = abcd$$

Tipp: Eine dieser Zahlen ist beispielsweise die 1233 ($= 12^2 + 33^2$)

2. Erstellen Sie ein Programm das eine Würfelstatistik erstellt. Dabei soll die Anzahl der benötigten Würfe vor der Ausführung eingelesen werden, die Ausgabe soll folgendes Aussehen haben (Beispiel für 25 Würfe):

Würfelstatistik

Bei 25 Würfeln entfielen

auf die 1: 2 Würfe

auf die 2: 6 Würfe

auf die 3: 5 Würfe

auf die 4: 6 Würfe

auf die 5: 2 Würfe

auf die 6: 4 Würfe

4.4 Funktionen

Ähnlich wie Schleifen werden Funktionen dazu benutzt einmal abgelegten Programmcode mehrfach zu benutzen. Während Schleifen jedoch automatisiert ablaufen, werden Funktionen manuell mit verschiedenen Initialisierungswerten gestartet. Nehmen wir zur Verdeutlichung einmal an, wir wollen von den Zahlenpärchen (2,5), (6,14) und (1048, 734) den Mittelwert berechnen. Mit unseren bisherigen JavaScript-Möglichkeiten sähe das wie folgt aus:

```
var sum1, sum2;
var ergebnis;

// Erstes Zahlenpärchen
sum1 = 2;
sum2 = 5;
ergebnis = (sum1 + sum2) / 2;
document.write("Der Mittelwert der Zahlen " + sum1 +
               " und " + sum2 " beträgt: " + ergebnis);

// Zweites Zahlenpärchen
sum1 = 6;
sum2 = 14;
ergebnis = (sum1 + sum2) / 2;
document.write("Der Mittelwert der Zahlen " + sum1 +
               " und " + sum2 " beträgt: " + ergebnis);

// Drittes Zahlenpärchen
sum1 = 1048;
sum2 = 734;
ergebnis = (sum1 + sum2) / 2;
document.write("Der Mittelwert der Zahlen " + sum1 +
               " und " + sum2 " beträgt: " + ergebnis);
...
```

Funktionen aufrufen

Sie werden mir beipflichten, dass die einzigen Änderungen in unserem zuletzt erstellten Programmcode in den ausgetauschten Zahlenwerten für die Variablen `sum` bestehen. Ziel unserer Anstrengungen soll es daher sein, mit Hilfe von Funktionen soviel Schreibarbeit wie möglich einzusparen. Hierzu werden wir im folgenden eine Funktion `mw` kreieren, so dass sich der eigentliche JavaScript-Code auf folgende Zeilen verkürzen lässt.

```
// Erstes Zahlenpärchen
document.write(mw(4,5));

// Zweites Zahlenpärchen
document.write(mw(6,14));
```

```
// Drittes Zahlenpärchen  
document.write(mw(1048,734));
```

Durch die oben aufgeführten Zeilen benutzen Sie schon etwas, was Sie bislang noch nicht besitzen, nämlich die Funktion `mw`. Wie Sie wahrscheinlich schon erkannt haben, übernimmt diese Funktion zwei *Argumente*, nämlich die zwei Zahlen deren Mittelwert berechnet werden soll.

Funktionen definieren

Die Definition einer Funktion erfolgt stets über das Schlüsselwort `function`, gefolgt vom *Funktionsnamen*. Es folgen in Klammern Variablennamen, welche die oben aufgeführten Argumente übernehmen sowie (in geschweiften Klammern) der eigentliche Funktionskörper oder -rumpf. Unsere Funktion `mw` nimmt dadurch folgende Gestalt an:

```
function mw (sum1, sum2) {  
  
    var summe = (sum1 + sum2) / 2;  
    var ergebnis = "<p>Der Mittelwert der Zahlen " + sum1 +  
                  " und " + sum2 + " beträgt: " + summe +  
                  "<p>";  
    return ergebnis;  
  
}
```

Beachten Sie sowohl bei dieser, als auch bei allen anderen Funktionen die folgenden Punkte:

- Die Anzahl der beim Aufruf übergebenen Argumente, muss der Anzahl der Variablen entsprechen, die beim Anlegen der Funktion vereinbart worden sind. Man spricht in diesem Zusammenhang auch von *Parametern*.
- Das Ergebnis der Funktion wird durch die Anweisung `return` an die Stelle des Funktionsaufrufs zurückgegeben. Die Angabe eines Variablennamen ist beim `return` nicht zwingend erforderlich, der Funktionswert wird dann als leer oder *void* bezeichnet.

Die Anweisung `return` kann auch während des Funktionsablaufs benutzt werden, die Funktion wird dann vorzeitig beendet. Fehlt das `return` ganz, so endet die Funktion in jedem Fall nach Ihrer letzten Anweisung.

Damit müssen Sie jetzt nur noch wissen, wo die Funktion in dem HTML-Dokument abgelegt wird, dann steht der Mittelwertberechnung nichts mehr im Wege.

Einbettung in die HTML-Seite

→ Abbildung 4.1

Prinzipiell können Sie Ihre Funktionen an jeder denkbaren Stelle in Ihr HTML-Dokument einfügen, die einzige formale Bedingung besteht darin, dass dem JavaScript-Interpreter die Funktion bekannt sein muss, *bevor* sie erstmalig aufgerufen wird. Aus diesem Grund ist es gute Tradition Funktionen im `<head>`-Bereich des HTML-Dokuments abzulegen, da dieser in jedem Fall vor dem `<body>`-Teil, in dem ja die eigentliche Verarbeitung stattfindet, gelesen wird. Das gesamte Mittelwertprogramm finden Sie in Abbildung 4.1.

```
1  <html>
2  <head>
3      <title>Mittelwertberechnung</title>
4      <script language="javascript">
5          <!--
6              function mw (sum1, sum2) {
7
8                  var summe = (sum1 + sum2) / 2;
9                  var ergebnis = "<p>Der Mittelwert der Zahlen " + sum1 +
10                      " und " + sum2 + " beträgt: " + summe + "<p>";
11                  return ergebnis;
12
13              }
14          //-->
15      </script>
16 </head>
17 <body>
18     <h3>Erster Mittelwert</h3>
19     <script language="javascript">
20         <!--
21             document.write(mw(4,5));
22         //-->
23     </script>
24     <h3>Restliche Mittelwerte</h3>
25     <script language="javascript">
26         <!--
27             document.write(mw(6,14));
28             document.write(mw(1048,734));
29         //-->
30     </script>
31 </body>
32 </html>
```

Abbildung 4.1: Das komplette Mittelwert-Script.

Variabel verwendbare Argumentlisten

Hoffentlich haben Sie erkannt, dass Sie durch die Verwendung von Funktionen viel Arbeit sparen können. Besonders dynamisch werden Funktionen jedoch, wenn die Anzahl ihrer Parameter flexibel gehalten wird. Als Beispiel dafür soll uns wieder die Mittelwertberechnung dienen, da sich Mittelwerte bekanntermaßen auch von mehreren Zahlen berechnen lassen. Gesucht wird daher eine Funktion, die wie folgt aufgerufen werden kann:

```
// Erster Mittelwert
document.write(mw(4,5));

// Zweiter Mittelwert
document.write(mw(4,5,6,14));

// Dritter Mittelwert
document.write(mw(4,5,6,14,1048,734));
```

Bislang haben wir jedes unserer Funktionsargumente auf einen vorgegebenen Parameter zugewiesen. Daneben wird jedes Argument aber beim Aufruf der Funktion in einem systemseitigen Array `arguments` hinterlegt. Dies werden wir nun beim Umbau der Funktion `mw` ausnutzen.

→ Abbildung 4.2 auf der nächsten Seite

Da wir die Argumentübergabe dieses Mal über das Array `arguments` ausführen, lassen wir die Parameterliste der Funktion leer. Das modifizierte Beispiel finden Sie in Abbildung 4.2

4.5 Variablen und ihr Gültigkeitsbereich

Gültigkeitsbereiche

Bislang haben wir Variablen mehr oder minder nach Gutdünken angelegt und benutzt, ohne uns großartig um einen Gültigkeitsbereich zu scheren – eventuell werden Sie sich sogar fragen, was das Wort Gültigkeitsbereich überhaupt bedeutet. Generell wollen wir darunter den Abschnitt eines HTML-Dokuments verstehen, in dem wir auf eine bestimmte Variable zugreifen können, das heisst den Bereich, in dem die Variable tatsächlich existiert. Verdeutlichen wir uns die ganze Thematik an einem Beispiel:

```
1  ...
2  <head>
3  <script language="javascript">
4  <!--
5      function funk () {
6          var x = 7;
7      }
8  //-->
9  </script>
10 </head>
```

```
1  <head>
2    <title>Mittelwertberechnung</title>
3    <script language="javascript">
4      <!--
5        function mw () {
6
7          var summe = 0;
8          var i = 0;
9          var anzahl = arguments.length;
10         var ergebnis = "<p>Der Mittelwert der Zahlen ";
11
12         for (i = 0; i < anzahl; i++) {
13
14           summe = summe + arguments[i];
15           ergebnis = ergebnis + arguments[i];
16
17           switch (i) {
18             case anzahl-1:
19               break;
20             case anzahl-2:
21               ergebnis = ergebnis + " und ";
22               break;
23             default:
24               ergebnis = ergebnis + ", ";
25           }
26
27         }
28
29         summe = summe / anzahl;
30         ergebnis = ergebnis + " beträgt: " + summe + "</p>";
31
32         return ergebnis;
33
34       }
35     //-->
36   </script>
37 </head>
```

Abbildung 4.2: Mittelwerte für beliebig lange Zahlenreihen.

```
11 <body>
12 <script language="javascript">
13 <!--
14     var x = 5;
15 //-->
16 </script>
17 ...
18 <script language="javascript">
19 <!--
20     document.write(x);
21     var x = 1045;
22     document.write(x);
23 //-->
24 </script>
25 ...
```

Ohne jetzt über den praktischen Nutzen des Beispiels streiten zu wollen, sollten Sie sich mit der Frage beschäftigen welchen Wert die Variable `x` zu welchem Zeitpunkt der Dokumentverarbeitung besitzt.

Die Antwort auf diese Frage ist relativ simpel. In Wahrheit haben wir es nicht mit einer sondern mit gleich *drei* Variablen zu tun, die zufälligerweise einen identischen Namen besitzen (nämlich `x`). Dass es dabei nicht zu Verwechslungen oder Überschneidungen kommt, verdanken wir dem Umstand, dass jede dieser Variablen `x` einen eng umfassten Gültigkeitsbereich besitzt. Für das `x` in der Funktion `funk` (Zeile 6) bedeutet dies, dass die Variable ausschließlich in der Funktion bekannt ist, direkt hinter dem abschliessenden „`}`“ kann auf die Variable, und damit auch auf den Inhalt, nicht mehr zugegriffen werden.

Für die Variablen in den beiden unteren `<script>`-Umgebungen gilt ähnliches. Hier endet die Gültigkeit mit dem Schließen des jeweiligen `<script>`. Dies bedeutet jedoch auch, dass die `document.write`-Anweisung in Zeile 20 einen Fehler erzeugen wird, da eine Variable `x` bis dahin nicht bekannt ist.

Globale Variablen

Eine Sonderrolle spielen in diesem Zusammenhang globale Variablen. Diese werden wie Funktionen im `<head>`-Bereich angelegt und sind damit in allen `<script>`-Umgebungen der HTML-Datei bekannt. Konstruieren wir uns auch diesmal eine entsprechende Testumgebung.

```
1 ...
2 <head>
3 <script language="javascript">
4 <!--
5     // Globale Variable x
6     var x = 64;
7
```



```

8      function funk () {
9          // Lokales x
10         var x = 7;
11     }
12     //-->
13 </script>
14 </head>
15 <body>
16 <script language="javascript">
17 <!--
18     // ??? x
19     document.write(x);
20     // Lokales x
21     var x = 1045;
22     document.write(x);
23     //-->
24 </script>
25 ...

```

Dieses Mal läuft unser JavaScript fehlerfrei, da die `document.write`-Anweisung in Zeile 19 in Ermangelung anderer Möglichkeiten auf das global in Zeile 6 definierte `x` zurückgreift. Erst danach wird eine lokale Variable gleichen Namens angelegt auf die weiterhin zugegriffen wird – ich hoffe Sie haben die Unübersichtlichkeit, die ein solcher Mix aus globalen und lokalen Variablen mit sich bringt, bemerkt.

→ Kapitel 3.8.2 auf Seite 26

Nichtsdestotrotz besteht auch nach dem Anlegen des *lokalen* `x` die Möglichkeit auf die globale Variable zuzugreifen. Wir bedienen uns dabei eines Schemas, welches Sie bereits aus dem Kapitel **Ausgaben** kennen.

Sie werden sich (hoffentlich) noch daran erinnern, dass wir bei den Ausgaben den Bezeichner `window` kennengelernt haben, über den wir auf Eigenschaften des Browserfensters (`document`, `status`) zugreifen können. Der Kunstgriff besteht nun darin zu wissen, dass alle *global* angelegten Variablen und Funktionen Eigenschaften von `window` sind und darüber auch in einer Form `window.Bezeichner` adressiert werden können. Wir sollten daher schon allein aus Übersichtlichkeitsgründen den unteren Teil des Scripts wie folgt ändern:

```

17 <!--
18     // Globales x
19     document.write(window.x);

```

4.6 Übungsaufgabe

Eine Schule in Göttingen um 1785. Ein Lehrer verlangt von seinen Schülern als Übung die Summe der Zahlen von 1 bis 100 zu berechnen. Aus dem Vorsatz den wohlverdienten freien Nachmittag genießen zu können wird jedoch nichts, da sich bereits nach kurzer Zeit der Schüler Gauß mit der richtigen Lösung meldet.

Aufgabe für den Rest des Nachmittags: Erstellen Sie eine JavaScript-Funktion `gausssum`, welche die Summe der Zahlen von 1 bis 100 errechnet. Die Funktion soll dabei in der Form

```
summe = gausssum(1, 100);
```

aufgerufen werden können, das heisst nach Ablauf der Funktion befindet sich in der Variable `summe` das gesuchte Ergebnis.

II Zweiter Tag

5 JavaScript und die HTML-Umgebung

Auch wenn sich noch sehr viel mehr zu Basiswissen und Kontrollstrukturen sagen lässt, so lassen wir es an dieser Stelle für den Moment genug sein. Schließlich hat dieses Skript den Anspruch Ihnen die Grundstrukturen von JavaScript zu vermitteln, und nicht die Sprache in ihrer Vollständigkeit. Beenden wir also die dumpfe Paukerei und betrachten wir die Dinge, in deren Umfeld der Einsatz von JavaScript erst richtig interessant wird. Dies ist in erster Linie die Kombination von JavaScript mit dem umgebenden HTML-Dokument.

Bislang haben Sie den HTML-Bereich wahrscheinlich eher als notwendiges Übel zur Kenntnis genommen; schließlich benötigen Sie einen JavaScript-Interpreter und dieser lässt sich scheinbar ohne eine HTML-Datei nicht benutzen.¹ Interaktionen mit den HTML-Elementen selbst hat es, sehen wir mal von dem Geplänkel mit der Ausgabe in das `document` oder die Statuszeile ab, bislang nicht gegeben. Dies soll sich jetzt ändern, denn schließlich ist es eine Hauptaufgabe von JavaScript, Bewegung in HTML-Dokumente zu bringen.

5.1 Ereignisse und Events

Bis jetzt sind wir stets von einem *linearen* Programmablauf ausgegangen, was erst einmal bedeutet, dass ein HTML-Dokument durch den Browser geladen wird. Innerhalb des Dokuments werden nun HTML-Anweisungen ausgeführt, darunter auch `<script>`-Umgebungen. Wie man es aber auch dreht und wendet, irgendwann wird die `</html>`-Marke erreicht und damit ist die Ausführung des Dokuments faktisch beendet.

Alles was nach diesem Zeitpunkt noch passieren kann, liegt im Ermessen des Benutzers. Das bedeutet: es können Fensterbereiche verändert werden, Tasten auf der Tastatur oder der Maus werden gedrückt, häufig wird lediglich der Mauszeiger bewegt. All diese Aktionen werden im Fachjargon *Ereignisse* oder *Events* genannt – da ja tatsächlich was Unvorhergesehenes passiert. Ihre Aufgabe als JavaScript-Programmierer besteht nun darin, auftretende Ereignisse zu erkennen („Gerade eben ist die Maus auf das Bild mit dem Haus bewegt worden“) und sie mit einer Reaktion zu verknüpfen („Es erscheint eine Meldung, dass das Haus schon verkauft ist“).

Dies mag jetzt arg kompliziert klingen, das Verfahren ist in Wirklichkeit aber simpler als es aussieht, da Sie von JavaScript sehr weitreichend

¹ In der Tat hat es tatsächlich mal Ansätze gegeben JavaScript auch als eigenständige Shell-Script-Sprache zu etablieren, letztendlich konnte sich die Sprache jedoch nicht gegen die etablierte Konkurrenz wie Perl, Python oder Tcl durchsetzen.

unterstützt werden. Der WWW-Browser stellt Ihnen zu den meisten Ereignissen einen sogenannten *EventHandler* zur Verfügung der sofort reagiert, sobald ein Event stattgefunden hat. Der Grund warum es auf Ihrem Computerbildschirm in aller Regel eher ruhig zugeht, ist der Tatsache zu verdanken, dass die meisten EventHandler standardmäßig mit einer Null-Aktion vorbelegt sind; das heisst, sie registrieren zwar jedes Ereignis führen daraufhin jedoch keine Aktion durch.

Überschreiben von Eventhandlern

Ziel unserer Vorgehensweise muss es also sein, das Verhalten eines Eventhandlers so zu ändern, dass er statt des bisherigen Nichtstuns eine von uns vorgegebene Aktion ausführt. Betrachten wir daher ein Beispiel:

```

```

→ Kapitel 3.8.2 auf Seite 27

Durch den ``-Tag wird ein Bild der FernUniversität in das HTML-Dokument gesetzt. Zusätzlich überschreiben wir den EventHandler `onMouseOver`, so dass in der Statuszeile des Browsers der Text *Eine Universität* erscheint, sobald die Maus über das Bild gezogen wird. Wir fassen zusammen:

- EventHandler werden vom WWW-Browser bereit gestellt und sind daher kein ausschließliches JavaScript-Feature. Sie müssen dem Browser daher bei der auszuführenden Aktion mitteilen, dass es sich im folgenden um JavaScript-Anweisungen handelt. Dies geschieht über den Zusatz `javascript:`²
- Der Text *Eine Universität* wird in diesem Fall mit *einfachen Anführungszeichen* (') gequotet, da die *doppelten* (") bereits für die Einbettung der JavaScript-Anweisung benutzt worden sind.

Zudem hat unser Beispiel noch den Schönheitsfehler, dass die Statuszeile durch die Mausbewegung zwar besetzt wird, sich der Inhalt jedoch nicht mehr ändert, wenn der Mauszeiger das Bild wieder verlässt. Abhilfe schafft das Überschreiben eines weiteren Eventhandlers.

```

```

² Tatsächlich gehen die meisten WWW-Browser intern davon aus, dass es sich um JavaScript-Anweisungen handelt, sollte der Zusatz `javascript:` einmal fehlen. Sie sollten sich auf dieses Verhalten aber nicht unbedingt verlassen.

Ereignisse weiterbehandeln

Ich verrate an dieser Stelle wohl keine großen Geheimnisse mehr, wenn ich behaupte, dass Eventhandler im überwiegenden Großteil aller HTML-Tags verwendet werden können. Ein zusätzliches Problem beschert uns jedoch der Eventhandler `onClick`.

`onClick` überprüft, ob auf dem aktuellen HTML-Tag eine Maustaste gedrückt worden ist. Dabei geraten wir bei Konstrukten wie dem folgenden jedoch in einen argen Konflikt.

```
<a href="http://www.wildeseiten.de"
  onClick="javascript:window.alert('Sie müssen mindestens 18 sein!!');">
Hier entlang zur aufregendsten Seite Ihres Lebens!</a>
```

Sofern es jetzt nicht nur bei Ihrer Maus geklickt hat, ist das Dilemma offensichtlich. Einerseits soll durch den Mausklick die JavaScript-Anweisung ausgeführt werden, andererseits liegt es in der Natur des Eventhandlers `onClick` bei einem `<a>`-Tag eine neue Web-Seite aufzurufen. Was wird daher tatsächlich passieren?

Die Antwort ist die naheliegendste. Der Eventhandler wird zuerst das JavaScript ausführen und danach die neue WWW-Seite laden. Ist dieses Verhalten nicht erwünscht so kann die Weiterverarbeitung, in unserem Fall der Aufruf der Seite, durch die Zusatzanweisung `return false` verhindert werden. Dieses Verhalten kann für einfache Sicherungsmechanismen ausgenutzt werden, wie sie uns das nächste Beispiel demonstrieren wird.

```
...
function checkAge () {

var weiter = true;

if (window.prompt("Geben Sie bitte Ihr Alter ein:", "") < 18) {
    weiter = false;
}

return weiter;
}

...

<a href="http://www.wildeseiten.de"
  onClick="javascript:return checkAge();">
Hier entlang zur aufregendsten Seite Ihres Lebens!</a>
```

Wenn wir mal davon absehen, dass wohl bereits die meisten 16jährigen diesen Sicherungsmechanismus knacken können, ist das Prinzip schon ganz brauchbar.

Häufig benutzte Eventhandler

Eventhandler existieren für beinahe alle vorstellbaren Ereignisse, allerdings sind nicht alle Teil des HTML-Standards (✓), sondern wurden häufig in Eigeninitiative der Browserhersteller hinzugefügt (✗). Die Verwendung der zuletzt genannten Sorte kann deshalb zu Problemen mit unterschiedlichen Browsern führen.

onLoad ✓

Nach Laden des Dokuments	<code><body onLoad="..."></code>
Nach Laden der Rahmen eines Framesets	<code><frameset onLoad="..."></code>
Nach Laden eines Bildes	<code></code>

onUnload ✓

Beim Verlassen von WWW-Seiten	<code><body onUnload="..."></code>
Schließen oder Ersetzen des Framesets	<code><frameset onUnload="..."></code>

onError ✗

Fehler beim Laden von Dokumenten	<code><body onError="..."></code>
Fehler beim Laden von Framesets	<code><frameset onError="..."></code>
Fehler beim Laden einer Grafik	<code></code>

onResize ✗

Größenänderung des Browserfensters	<code><body onResize="..."></code>
------------------------------------	--

onMove ✗

Verschieben des Browserfensters	<code><body onMove="..."></code>
---------------------------------	--

onMouseOver ✓

Mauszeiger bewegt sich in einen Bereich	<code><a href="..."</code> <code>onMouseOver="..."></code> <code><img src="..."</code> <code>onMouseOver="..."></code>
---	---

onMouseOut ✓

Mauszeiger bewegt sich aus einem Bereich	<code><a href="..."</code> <code>onMouseOut="..."></code> <code><img src="..."</code> <code>onMouseOut="..."></code>
--	---

onClick ✓

Anklicken eines Elements	<pre> <input type="button" onClick="..."> <input type="reset" onClick="..."> <input type="submit" onClick="..."> <input type="checkbox" onClick="..."> <input type="radio" onClick="..."> </pre>
--------------------------	---

onDbClick ✓

Doppelklick auf ein Element	siehe onClick
-----------------------------	---------------

onKeyPress ✓

Tastaturanschlag	<pre> <input type="text" onKeyPress="..."> <textarea onKeyPress="..."> <img src="..." onKeyPress="..." </pre>
------------------	--

onFocus ✓

Cursor wird in einem Eingabeelement positioniert (Fokussierung)	<pre> <input type="button" onFocus="..."> <input type="reset" onFocus="..."> <input type="submit" onFocus="..."> <input type="checkbox" onFocus="..."> <input type="radio" onFocus="..."> <input type="text" onFocus="..."> <textarea onFocus="..."> <select onFocus="..."> </pre>
---	--

onBlur ✓

Element wird der Fokus entzogen	siehe onFocus
---------------------------------	---------------

onChange ✓

Änderungen des Inhalts und anschließendes Verändern des Fokus	<pre> <input type="text" onChange="..."> <textarea onChange="..."> <select onChange="..."> </pre>
---	---

onReset ✓

Formulareingaben
werden durch
zurückgesetzt

```
<form action="..."
onReset="..."
```

onSubmit ✓

Formulareingaben wer-
den durch
abgesendet

```
<form action="..."
onSubmit="..."
```

Spezifizieren von Events

Mit den oben gezeigten Möglichkeiten besitzen wir ein mächtiges Werkzeug, um auftretende Ereignisse abzufangen und zu verarbeiten. Ein Schwachpunkt bleibt jedoch, dass wir lediglich die *Art* des auftretenden Events lokalisieren können. So tritt der Eventhandler `onClick` bei jeder Betätigung einer Maustaste in Aktion, was bei Macintosh-Geräten noch problemlos ist, unter Unix-Betriebssystemen die dort vorhandenen Möglichkeiten jedoch eindeutig reduziert.³ Noch offener tritt das Problem beim Eventhandler `onKeyPress` auf, bei dem es ja wirklich interessant werden kann, *welche* der im Durchschnitt 105 vorhandenen Tasten gerade gedrückt worden ist.

Abhilfe schafft das JavaScript-Ereignisobjekt `event`, welches jedoch erst seit der Version 1.2 existiert – Besitzer älterer WWW-Browser sollten sich daher vorsehen.⁴ Im nächsten Beispiel nutzen wir `event` um in einem Textfeld lediglich Zahleneingaben zuzulassen.

```
<form>
<input type="text"
      onKeyPress="javascript:if ((event.which > 57) ||
                                (event.which < 48)){return false};">
</form>
```

Das zu dem Objekt `event` gehörende Attribut `which` kann dabei folgende Werte enthalten. Bei

Tastaturbetätigung : den ASCII-Dezimalwert der gedrückten Taste. So erzeugt beispielsweise die Taste 0 den ASCII-Wert 48.

Mausklick : die numerischen Werte 1 (linke Maustaste), 2 (mittlere Maustaste), oder 3 (rechte Maustaste).

Weitere zu `event` gehörende Attribute:

³ Für die Betriebssystem-Laien zur Erklärung: Aufgrund des überragenden Designs der Benutzungsoberfläche, kommen Apple-Betriebssysteme mit einer einzigen Maustaste aus. Unix-Anwender müssen sich dagegen gleich mit drei Maustasten, und den damit verbundenen unterschiedlichen Events auseinandersetzen.

⁴ Auch wenn ich mich selbst eher zu den Software-Traditionalisten zählen würde, irgendwann kommt auch mal der Zeitpunkt um alte Zöpfe endgültig abzuschneiden.

<code>modifiers</code>	gibt an, ob zusätzlich <code>Ctrl</code> , <code>Alt</code> oder <code>Shift</code> gedrückt wurde
<code>pageX</code>	relative horizontale Position auf der Seite
<code>pageY</code>	relative vertikale Position auf der Seite
<code>target</code>	Art des Objekts, welche das Ereignis ausgelöst hat
<code>type</code>	Zeichenkette für die Art des Ereignisses
<code>which</code>	Dezimalwert für die Tastatur oder Maustaste

5.2 Weitere Dokumente laden

Eine häufig im Zusammenhang mit Eventhandlern gebrauchte Eigenschaft ist es, neue Dokumente in das aktuelle Browserfenster zu laden. Der Generalschlüssel hierzu ist wieder mal das Objekt `window`, bzw. dessen Attribut `location`, in dem jeweils die Adresse der gerade „aktiven“ Seite hinterlegt ist. Durch gezieltes Überschreiben dieses Attributs kann der Fensterinhalt verändert werden.

Die Eigenschaft `window.location.href`

In Analogie zur Besetzung der `status`-Eigenschaft, bewirkt die Belegung von `href` mit einem gültigen URL, die Anzeige der Seite die sich hinter dieser Adresse verbirgt. So sind Sie beispielsweise durch eine JavaScript-Zeile in der Lage, ein Datei→Öffne WWW-Adresse-Fenster zu simulieren.

```
<script language="javascript">
  <!--
    window.location.href=
      window.prompt("Geben Sie eine gültige Adresse ein");
  //-->
</script>
```

Die Funktion `replace`

Ein scheinbar analoges Verhalten zeigt die ebenfalls in `location` enthaltene Funktion `replace`. Unser Beispiel ließe sich daher auch so schreiben:

```
<script language="javascript">
  <!--
    window.location.replace(
      window.prompt("Geben Sie eine gültige Adresse ein"));
  //-->
</script>
```

Im Gegensatz zum Überschreiben der `href`-Eigenschaft, löscht `replace` jedoch den ursprünglichen Eintrag in der History Ihres WWW-Browsers.

Sie haben dadurch erstklassige Möglichkeiten die Seitenverlaufanzeige zu verwischen, wie es das folgende Beispiel einer Weiterleitung zeigt, ohne dass die ursprüngliche Seite einen Eintrag in der History des Browsers hinterlässt.

```
<body onLoad="window.location.replace('http://www.weiter.de');">
<p>Eine Textzeile die nie jemand lesen wird...
```

Der Pferdefuß lässt in diesem Zusammenhang nicht lange auf sich warten, da bekanntlich alle positiv verwertbaren Werkzeuge auch in einem negativen Sinn angewandt werden können. Gewissermaßen als Denksportaufgabe sollten Sie den Sinn des folgenden HTML-Dokuments herausbekommen – spätestens das praktische Ausprobieren sollte die gewünscht nachhaltige Wirkung erzielen.

```
<body onUnload="window.location.replace(window.location.href);">
<h1>Trapped!!!</h1>
```

Die Funktion open

Sofern nicht das komplette Browserfenster, sondern lediglich ein darin enthaltener Frame überschrieben werden soll, bietet sich als dritte Alternative die `window`-Funktion `open` an, deren allgemeine Form sich mit

```
window.open(URL, Frame)
```

beschreiben lässt. Anstelle eigener Frames können auch üblichen Standard-Frames verwendet werden.

<code>_self</code>	ersetzt das aktuelle Fenster, bzw. den aktuellen Frame
<code>_parent</code>	überschreibt den Frameset, in dem der aktuelle Frame definiert ist
<code>_top</code>	ersetzt den Frameset der obersten Hierarchiestufe
<code>_blank</code>	erzeugt ein neues Fenster

5.3 Eigene Fenster aufbauen

Eine weitere Eigenschaft von JavaScript, die, vorsichtig formuliert, nicht immer erwünscht ist, besteht in der Möglichkeit neue Fenster zu erzeugen und auf dem Bildschirm zu platzieren. Diese Unart der Popup-Überflutung wird dabei nicht nur von unseriösen Anbietern genutzt, auch die deutsche Telekom sowie diverse Internet-Provider wenden sie gelegentlich gern an.

Basis dieser Technik ist die im letzten Abschnitt vorgestellte Funktion `window.open`, diesmal jedoch mit modifizierter Argumentliste.

```
window.open("", "", Optionen)
```

An den beiden ersten leeren Argumenten erkennt der JavaScript-Interpreter, dass ein neues Fenster geöffnet werden soll, welches über weitere *Optionen* layoutiert werden kann. Die Optionenliste darf folgende Werte enthalten:

Option	Mögliche Werte	Bedeutung
menubar	yes/no	Anzeige der Browsermenüleiste
toolbar	yes/no	Anzeige der Navigations-schaltflächen
directories	yes/no	Schaltflächen für Favoriten-verzeichnisse
location	yes/no	Anzeige/Eingabefeld für die Adresszeile
status	yes/no	Anzeige der Statuszeile
scrollbars	yes/no	Anzeige der Bildlaufleisten
resizable	yes/no	Fenstergröße darf geändert werden
width	Pixel	Breite des Fensters
height	Pixel	Höhe des Fensters

! → Einige exotischere WWW-Browser verlangen die Angabe von 1 und 0 anstelle von yes und no. Im Zweifelsfall hilft da nur ausprobieren.

→ Beispiel 5.1 auf der nächsten Seite


Um unsere Kenntnisse in aktivem Spamming zu vertiefen, bauen wir nun ein Dokument mit automatischem Spam-Fenster. Als Vorlage können Sie das JavaScript-Listing auf Seite 58 nehmen.

Beachten Sie dabei, dass wir die Referenz auf das in Zeile 16 geöffnete Fenster in der globalen Variable `spamfenster` speichern (Zeile 6) – auf den Grund dieser Maßnahme werden wir zu einem späteren Zeitpunkt zurückkommen.

Nach dem Öffnen wird das Fenster auf dem Bildschirm durch die Anweisung `moveTo` positioniert, der Rest ist die Ausgabe von (beinahe) reinem HTML-Code. Selbstverständlich dieser auch wieder JavaScript-Code enthalten, unter anderem lassen sich alle Eventhandler verwenden, die uns bislang bekannt sind. Wir könnten beispielsweise beim Schließen des Fensters (Stichwort `onUnload`) ein neues Werbefenster öffnen, welches wiederum... aber das sind wohl reine Phantasiegespinste.

Da wir indes freundliche Spammer sind, wollen wir das Fenster wieder schließen, sobald die rufende Seite entladen wird. Dabei stoßen wir allerdings auf ein weiteres Problem. Prinzipiell könnte das Fenster durch die Anweisung `window.close()` eliminiert werden, wir haben jedoch tatsächlich keine Kontrolle darüber, ob das Fenster zu diesem Zeitpunkt überhaupt noch aktiv ist.⁵ Zudem reagieren viele JavaScript-Interpreter neurotisch, wenn Fensteroperationen auf nicht mehr existierende Fenster angewandt werden. Abhilfe schafft die Eigenschaft `closed`, die als

→ Beispiel 5.2 auf Seite 59

⁵ Viele Benutzer reagieren auf PopUps nämlich mit einem nervösen Mausklick auf das -Symbol oben rechts.

```
1  ...
2  <head>
3  <script language="javascript">
4  <!--
5  //Referenz auf das neue Fenster
6  var spamfenster;
7
8  //Funktion zum Aufbau
9  function zeigeSpam() {
10
11  var optionen = 'menubar=no,toolbar=no';
12  optionen=optionen + ',directories=no,location=no';
13  optionen=optionen + ',status=no,scrollbars=no';
14  optionen=optionen + ',width=360,height=180';
15
16  spamfenster = window.open("", "", optionen);
17  spamfenster.moveTo(50, 50);
18  spamfenster.document.write("<html><body><h1>");
19  spamfenster.document.write("Hier könnte IHR Spam stehen!".fontcolor("red"));
20  spamfenster.document.write("</h1></body></html>");
21
22  }
23  //-->
24  </script>
25  </head>
26  <body onLoad="javascript:zeigeSpam();">
27  ...
```

Abbildung 5.1: Die allseits beliebten PopUp-Fenster können mit JavaScript leicht realisiert werden. Mit den oben stehenden Anweisungen bauen Sie beispielsweise eine Werbefläche auf, die Sie leicht vermieten können ;-)

boolschen Wert den momentanen Zustand des Fensters beinhaltet. Wir erweitern unser Beispiel daher so, wie im Listing auf Seite 59 ersichtlich.

Gleichzeitig lösen wir auch die Frage, warum die Variable `spamfenster` global angelegt worden ist: weil wir sie später noch in der Funktion `zeigeKeinenSpam` benötigen, um das Fenster wieder sauber zu deaktivieren.

5.4 HTML-Objekte

Erinnern Sie sich noch an Kapitel 1.2? Sofern Sie wie ich zu den Menschen gehören die ein schlechtes Zahlengedächtnis besitzen: Zu Beginn dieses Kapitels lernten wir, dass JavaScript „eine Sprache mit rudimentärer Objektorientierung“[\[1\]](#) ist.

```
1  ...
2  <head>
3  <script language="javascript">
4  <!--
5  //Referenz auf das neue Fenster
6  var spamfenster;
7
8  //Funktion zum Aufbau
9  function zeigeSpam() {
10  ...
11  }
12
13  //Funktion zum Abbau
14  function zeigeKeinenSpam() {
15
16      if (!spamfenster.closed) {
17          spamfenster.close();
18      }
19
20  }
21  //-->
22  </script>
23  </head>
24  <body onLoad="javascript:zeigeSpam();"
25      onUnload="javascript:zeigeKeinenSpam();">
26  ...
```

Abbildung 5.2: PopUp-Fenster lassen sich programmtechnisch sehr einfach entfernen – leider nimmt dies kaum ein ernsthafter Spammer zur Kenntnis.

Gerade aufgrund dieser Einschränkung⁶ habe ich bislang versucht Ihnen die Objektorientierung vom Hals zu halten. Allein bei diesem Kapitel bemerke ich einen leisen Zweifel, dass es ganz ohne dann doch nicht geht. Schieben wir also einen Schnellkurs in Bezug auf Objektorientierung ein.

Begriffe aus dem Objektfeld, eher allgemeinverständlich

Lösen wir uns für einen Moment von unserem JavaScript- und auch sonstigem IT-Umfeld, und betrachten wir irgendein *Ding* in unserer Umgebung, beispielsweise eine Kuh.⁷ Sie werden mir beipflichten, dass sich diese Kuh ganz allgemein nicht nur als Ding sondern auch als *Objekt* bezeichnen lässt.

⁶ Für zukünftige Versionen dieser Broschüre wäre ich dankbar, wenn mir jemand bei Gelegenheit das Substantiv von „rudimentär“ nennen könnte.

⁷ Sofern in Ihrer Umgebung just zu diesem Zeitpunkt keine Kuh greifbar ist, sollte Sie nichts daran hindern sich eine vorzustellen.

Betrachten wir diese Kuh etwas näher (Vorsicht! Nicht zu nah, und alles Rote vorher ausziehen), so stellen wir fest, dass eine Kuh unter anderem aus Augen, Ohren, Hörnern, einem Schwanz, sowie Beinen besteht. Wir bezeichnen diese Eigenschaften der Kuh im Fachjargon auch als ihre *Attribute*. Da auch ein Ohr eine Eigenschaft haben kann, vielleicht ist es ein braunes Ohr, dürfte somit auch klar sein, dass Attribute ebenfalls Objekte sein können, die weitere Attribute besitzen. Soweit klar?

Gut, dann spendieren wir unserer Kuh ein Färbemittel und lackieren ihr Ohr in violett. Anders ausgedrückt, besetzen wird das Attribut *Farbe*, des Objekts *Ohr*, welches wiederum ein Attribut des Objekts *Kuh* ist, mit *lila*, oder kurz geschrieben:

```
Kuh.Ohr.Farbe='lila';
```

Sofern Ihnen diese Schreibweise bereits bekannt vorkommt, sind Sie schon auf dem richtigen Weg. Es geht aber noch weiter: Da unsere Kuh äusserst kitzelig ist, kann ihre Ohrfärbung nur mit einer Sprühdose geändert werden; jeder Versuch dies mit Pinseln zu tun führt unweigerlich zu Huftritten. Die Lösung besteht darin, dass wir in unserem Objekt *Ohr* eine Färbefunktion *faerbeOhr(neuefarbe)* definieren, die für uns die Ohren der Kuh färbt und gleichzeitig darauf achtet, dass dies mit einer Sprühdose geschieht.

```
Kuh.Ohr.faerbeOhr('lila');
```

Solche Objektfunktionen heissen auch *Methoden*, und dabei wollen wir es dann auch belassen. Vielleicht eines noch: Diese Broschüre wurde *nicht* von einer bekannten Schokoladenfirma finanziert.

Das Objektfeld von JavaScript

Sie haben hoffentlich spätestens jetzt erkannt, dass wir bereits ziemlich heftig mit Objekten gedealt haben. Das bekannteste unter ihnen ist wohl das Objekt *window*, von dem wir schon einige Attribute (*document*, *status*) als auch Methoden zur Änderung der Attributwerte (*write*, *moveTo*) kennengelernt haben. Speziell durch das Sub-Objekt *document* haben wir nun die Möglichkeit, den Inhalt von HTML-Dokumenten gezielt zu manipulieren.

Der JavaScript-Interpreter ist nämlich in der Lage, die auf der Dokumentenseite benutzten Elemente in eigens dafür vorgesehenen Arrays zu verwalten; Abbildung 5.3 zeigt den hierarchischen Aufbau.⁸ Gehen wir einmal von dem folgenden, minimalistischem HTML-Dokument aus:

→ Abbildung 5.3 auf der nächsten Seite

⁸ Neben den gezeigten Element-Arrays werden zusätzlich die Bereiche *anchors* und *applets* verwaltet. Da diese aber generell in der Praxis und speziell in dieser Broschüre keine Rolle spielen, habe ich auf die Auflistung aus Gründen der Übersichtlichkeit verzichtet.

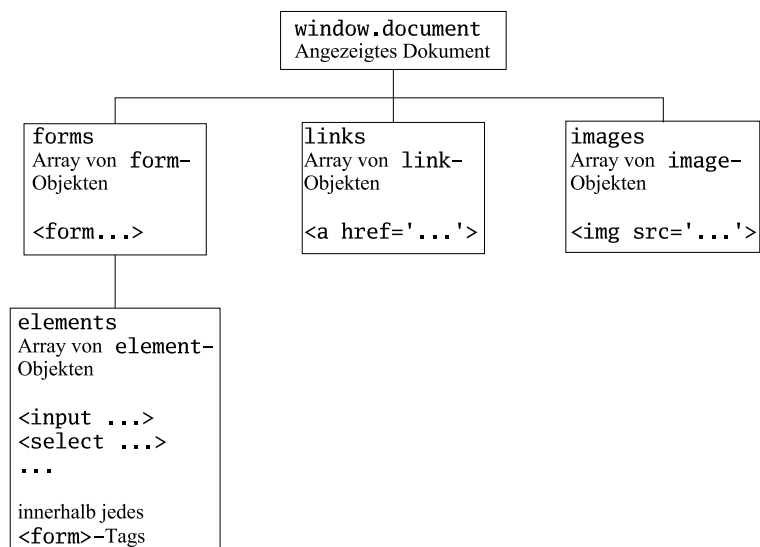


Abbildung 5.3: Über die Arrays der document-Objekte, haben Sie Zugriff auf viele HTML-Elemente.

```

<html>
<body>

<h1>Eine Überschrift</h1>
<p>Ein wenig <a href="neuerText.html">Text</a> und ein erläuterndes
Bild </p>
</body>
</html>
  
```

Von der JavaScript-Seite aus betrachtet, besitzen nun mindestens die Arrays `window.document.images`, sowie `window.documents.links` Elemente, wobei die Elementpositionen der Reihe nach vergeben werden. So beinhaltet das Element `window.document.images[0]` eine Referenz auf das Bild `logo.gif`, der Bildname kann über das Attribut `src` abgefragt werden.

Da wir mittlerweile auch die Erfahrung gemacht haben, dass sich Array-Elemente nicht nur lesen, sondern im Umkehrfall auch besetzen lassen, könnten wir mit der JavaScript-Anweisung

```

window.document.links[0].href="andererText.html"
  
```

dem im Text vorhandenen Link ein ganz neues Sprungziel zuweisen. HTML-Dokumente können dadurch eine gewisse Eigendynamik entwickeln, wie das nächste Beispiel zeigen wird.

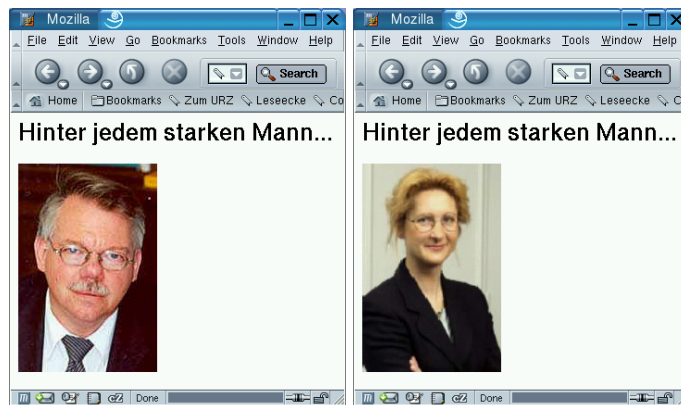


Abbildung 5.4: Hinter jedem starken Mann... steckt bekanntlich immer eine starke Frau.

```

1  <html>
2  <head>
3  <script language="javascript">
4  <!--
5  //Referenz auf das alte Bild
6  var bildref;
7  //-->
8  </script>
9  </head>
10 <body>
11   <h2>Hinter jedem starken Mann...</h2>
12   
16 </body>
17 </html>

```

→ Abbildung 5.4

Der Austausch des Bildes, im Fachjargon auch *Roll-Over-Effekt* genannt, ist dabei denkbar einfach. Sobald die Maus das ursprüngliche Bild berührt, wird dessen Name über das Attribut

```
window.document.images[0].src
```

in der globalen Variable `bildref` gesichert; das selbe Attribut wird zum gleichen Zeitpunkt mit dem Namen des neuen Bildes überschrieben. Um den Effekt rückgängig zu machen (`onMouseOut`) reicht es, das Attribut mit dem gesicherten Bildnamen zu überschreiben.

! → Prinzipiell hätte es im oberen Beispiel auch gereicht, das Attribut `window.document.images[0].src` mit dem festen Dateinamen zu besetzen. Da Rektoren an dieser Hochschule jedoch häufiger als KanzlerInnen wechseln, haben wir mit dem gezeigten Verfahren schon mal für zukünftige Weichenstellungen vorgesorgt ;-)

Sobald Sie sich von dem sicherlich überwältigenden Eindruck erholt haben, will ich auf eine Schwäche des gezeigten Verfahrens eingehen. Dadurch, dass die Array-Elemente von `images` in der Reihenfolge der auftretenden Grafiken besetzt werden, ist unser Bildaustausch zum Scheitern verurteilt, sobald der erste Web-Designer beispielsweise ein Logo im Kopf der Seite einfügt.⁹

Dieses Hindernis lässt sich jedoch umgehen, indem bei dynamisch verwendeten HTML-Elementen das Attribut `name` benutzt wird, worüber eine eindeutige Zuordnung ermöglicht wird. Unser Beispiel wird folgerichtig modifiziert:

```

1      <h2>Hinter jedem starken Mann...</h2>
2      

```

Die Methode `getElementById`

JavaScript-Interpreter, die das neuere *Document Object Model* (DOM) unterstützen,¹⁰ können über die Methode `getElementById` auf sämtliche Elemente eines HTML-Dokuments zugreifen, sofern diese vorab durch das Attribut `id` eindeutig benannt worden sind.

Wir können daher nicht nur Bilder austauschen, auch ein Modifizieren der Überschrift ist ab sofort möglich; der eigentliche Inhalt wird im Attribut `innerHTML` vorgehalten.

```

1      <h2 id="phrase">Hinter jedem starken Mann...</h2>
2      

```

⁹ Die Funktionsfähigkeit bliebe zwar voll erhalten, jedoch würde *das Logo* durch das Bild der Kanzlerin ersetzt. Auch wenn ein solches Verfahren durchaus seinen eigenen Reiz hat, so ist es bestimmt nicht erwünscht.

¹⁰ Momentan behaupten zumindest die großen Hersteller Microsoft, Netscape und Mozilla, dass ihre *aktuellen* Browser dazu in der Lage sind. Ob, und in welchem Umfang dies tatsächlich der Fall ist, lässt sich häufig nur durch Ausprobieren rauskriegen.

! → Noch einmal zur Erinnerung: Der Zugriff auf HTML-Elemente über DOM bietet vielfältige Möglichkeiten, ist aber gleichzeitig stark browserabhängig!

5.5 Übungsaufgaben

Erstellen Sie eine HTML-Seite, welche die Stationsanzeige eines Radios simuliert wie sie in der unten befindlichen Abbildung zu sehen ist. Die drei Stationsknöpfe sind mit den Sendern *Eins Live*, *Radio Luxemburg* und *Antenne Bayern* vorbelegt.

Durch Mausklick auf eine der Stationstasten soll der Name des voreingestellten Senders angezeigt werden.



Hilfestellung: Die Stationstasten werden durch ein zusammenhängendes Feld von Radio-Button realisiert, beispielsweise in der Form

```
<input type="radio" name="tasten">Station 1
<input type="radio" name="tasten">Station 2
```

Auf der JavaScript-Seite wird dadurch ein *zweidimensionaler Array* `window.document.forms[].elements['tasten'][]` aufgebaut. Ob ein Radio-Button-Element aktiviert ist, lässt sich über die Eigenschaft `checked` abfragen.

Beispiel:

```
window.document.forms[].elements['tasten'][1].checked
```

enthält ein `true`, sofern Stationstaste *zwei* aktiviert ist.

Die zugehörige Eingabezeile, die in unserem Beispiel eher als Ausgabezeile fungiert und den Namen `station` tragen soll, lässt sich über die Eigenschaft

```
window.document.forms[].elements['station'].value
```

besetzen.

Zusatz: Erweitern Sie das Programm, so dass Sie die voreingestellten Stationen durch eigene Sendernamen erweitern können. Dazu tragen Sie einen neuen Sender in das Namensfeld ein und klicken im Anschluss auf die Stationstaste, unter welcher der neue Sender zukünftig abgerufen werden soll.

5.6 Interaktion mit dem WWW-Browser

Interaktion ist in diesem Zusammenhang ein vielleicht etwas übertriebener Begriff, da die Aktion in aller Regel recht einseitig ausfällt. JavaScript fragt, der WWW-Browser antwortet, der umgekehrte Weg (JavaScript besetzt Eigenschaften des Browsers) bleibt in aller Regel versperert.

Informationen über den Browser

Über das Attribut `window.navigator` können Sie Informationen über den momentan verwendeten WWW-Browser abrufen, beispielsweise um bei komplexen HTML-Dokumenten ein unterschiedliches Verhalten zwischen verschiedenen Browsern auszugleichen. Die folgenden Eigenschaften können Sie *lesen* benutzen:

Attribut	Beschreibung	Anmerkung
<code>appName</code>	Anwendungsname des verwendeten Browsers	Mozilla outet sich als Netscape
<code>appCodeName</code>	Programmname des verwendeten Browsers	Netscape und der IE(!) outen sich als Mozilla
<code>appVersion</code>	Versionsnummer des Browsers	teilweise werden Informationen zur grafischen Oberfläche mitgegeben
<code>language</code>	die vom Benutzer eingestellte Sprachunterstützung	
<code>platform</code>	das als Basis verwendete Betriebssystem	
<code>userAgent</code>	Der String, der vom Browser als User-Agent-Header an den WWW-Server übergeben wird	meist ein Mix aus <code>appCodeName</code> , <code>appVersion</code> und <code>platform</code>

Cookies

Cookies sind die einzige Möglichkeit für einen WWW-Browser, Informationen auf Ihrem Rechner abzulegen, um sie zu einem späteren Zeitpunkt wieder zu verwenden. Im Gegensatz zu einer weit verbreiteten Ansicht ist es jedoch *nicht möglich* einen PC durch Cookies mit Viren und Würmern zu infizieren.¹¹ Cookies werden vielmehr dazu eingesetzt, ein Benutzerprofil über den Anwender zu erstellen, um ihn bei Bedarf, nun sagen wir mal unmittelbar, mit den für ihn interessanten Informationen zu versorgen.

Lösten Cookies bei ihrer Einführung durch Netscape noch heftige Proteste aus, da es bei den damaligen Browsern keine Möglichkeit gab, die

¹¹ Tatsächlich gehen mindestens 85% aller „Infektionsschäden“ auf das Konto von MS-Outlook-Benutzern, die davon überzeugt waren, dass sich tatsächlich ein Supermodel in sie verlieben wird, wenn sie einen bestimmten Anhang in ihrer Mailbox öffnen.

Annahme zu verweigern, so können Sie bei der heutigen Browser-Generation sehr diffizile Vorkehrungen treffen, welche Cookies letztendlich auf Ihrer Festplatte gespeichert werden.

JavaScript speichert alle für eine HTML-Seite relevanten Cookies im Attribut `window.document.cookie`, welches in der Form *name=wert* besetzt wird. Beispiel für den Cookie `meinKeks`:

```
window.document.cookie='meinKeks=Kruemel';
```

Werden für eine Seite mehrere Cookies gesetzt, so gestaltet sich die weitere Verarbeitung schwieriger, da die Cookies, durch ein *Semikolon* (;) getrennt, als Zeichenkette hintereinander geschrieben werden. Die Ausgabe

```
alert(window.document.cookie);
```

könnte ein

```
meinKeks=Kruemel; meinSaft=Kirsch; meinKaffee=kalt
```

zur Folge haben. Es muss daher ein wenig mehr Aufwand betrieben werden, um einen einzelnen Cookie aus dieser Liste zu extrahieren und auszuwerten. Das folgende Beispiel zählt die Aufrufe einer HTML-Seite, in dem es pro Aufruf einen Zähler um 1 erhöht und diesen als Cookie speichert.

```

1  <h2>Wie oft noch?</h2>
2
3  <p> Sie haben diese Seite doch bereits
4    <script language="javascript">
5      <!--
6        // splittet bei einem ; mit evtl. folgenden Leerzeichen
7        var cookiear = window.document.cookie.split(/; */);
8        var counts = 0;
9
10       for (var i=0; i < cookiear.length; i++) {
11         if (cookiear[i].substr(0,11) == 'PageCounter') {
12           counts = cookiear[i].substr(12);
13         }
14       }
15
16       window.document.cookie='PageCounter=' + ++counts;
17       document.write(counts + 'x');
18     </script>
19   </p>
20 besucht...</p>
```

Ein noch zu lösendes Problem besteht darin, dass die „Lebenszeit“ des Cookies mit dem Schließen des WWW-Browsers abgelaufen ist; beim nächsten Aufruf der Seite geht es folgerichtig wieder mit 1 los.

Ist dies nicht gewünscht, und als professioneller Profilersteller wünschen Sie dies garantiert nicht, muss der Cookie über den Schlüssel `expires` mit einem Verfallsdatum versehen werden. Dieses muss wiederum zwingend als GMT-Format formatiert sein.

Um die Lebensdauer unseres Cookies zu erhöhen ändern wir Zeile 17 in

```
window.document.cookie='PageCounter=' + ++counts +  
    '; expires=Wednesday, 31-12-03 23:59:59 GMT';
```

Browserverlauf

JavaScript kann über das Objekt `window.history` auf die Liste der von Ihnen besuchten WWW-Seiten zurückgreifen; diese sind unter anderem in den Attributen `current`, `next` und `previous` gespeichert. Aus Sicherheitsgründen ist Ihnen der Zugriff auf diese Attribute jedoch nicht gestattet, so dass sich Ihre Eingriffe in die Browser-History auf die folgenden Methoden beschränken:

<code>back()</code>	springt zurück auf die zuletzt besuchte Adresse
<code>forward()</code>	springt vorwärts auf eine zuvor besuchte Seite
<code>go(n)</code>	führt für $n > 0$ n mal die Funktion <code>forward</code> aus. Für $n < 0$ wird entsprechend häufig die Methode <code>back</code> aufgerufen.

→ Kapitel 5.2 auf Seite 55

Denken Sie daran, dass mit der Funktion `location.replace()` ersetzte Seiten keinen Eintrag in der History erzeugen und daher durch die oben aufgeführten Methoden nicht mehr erreichbar sind.

5.7 Übungsaufgaben

1. Erweitern Sie das im letzten Übungsblock erstellte Dampfradio, so dass der unter *Station1* eingestellte Sender auch

- nach einem Neuladen der HTML-Seite, oder
- nach einem Neustart des WWW-Browsers

erhalten bleibt.

2. Erstellen Sie eine HTML-Datei in der ein Bild der FernUni angezeigt wird (das Bild erhalten Sie von Ihrem Übungsleiter). Durch Mausklick auf das Bild soll eine vergrößerte Darstellung des Bildes in einem eigenen Fenster angezeigt werden.

Tipp: Sie benötigen für die Vergrößerung kein zweites Bild. Weisen Sie dem neuen Fenster die Abmessungen 720x420 Pixel zu, und besetzen Sie in der darin enthaltenen ``-Marke den Parameter `width='100%'`.

6 Erweiterte Techniken

Gratulation, wenn Sie bis hierhin durchgehalten haben, so haben Sie den Grundparcours mit Erfolg absolviert – ich gehe zumindest mal davon aus, dass dem so ist. Die noch folgenden Kapitel zeigen Ihnen demnach auch nichts weltbewegend Neues mehr, es geht mehr darum bereits erlernte Techniken in eventuell anderen Zusammenhängen anzuwenden, um häufiger auftretende Probleme zu lösen.

6.1 Zeitgeber

Es mag etwas verwirrend klingen, aber Zeitgeber werden immer dann benötigt, wenn auf einer HTML-Seite automatisiert Bewegung erzeugt werden soll. Als Paradebeispiele für diese Art von Bewegungsanforderungen seien hier Bilderanimationen und Textlaufleisten genannt. Nehmen wir uns daher die Zeit, JavaScript einen Zeitgeber unterzuschieben.

Kernstück unserer künftigen Animationen ist dabei die Methode `window.setTimeout()` in der Form

```
ID = window.setTimeout("Anweisungen", pause);
```

Diese Methode arbeitet so, dass sie nach ihrem Aufruf die gesamte Verarbeitung *pause* Millisekunden anhält, *bevor* die JavaScript-Anweisungen ausgeführt werden.¹ Als Ergebnis gibt `setTimeout` ein numerisches Auftragssticket *ID* zurück. Bevor der nächste zeitgesteuerte Auftrag gestartet wird, sollten abgelaufene Aufträge durch Verwendung der Funktion `clearTimeout(ID)` aus dem Speicher entfernt werden.

→ Beispiel 6.1 auf der nächsten Seite

Wie nun durch geschicktes Stoppen und Starten zeitgesteuerter Aufträge eine Bildershow erzeugt wird, zeigt das nächste Beispiel, in dem zwei Bilder im Sekundentakt getauscht werden.

Schauen wir uns das Ganze in der Reihenfolge des Ablaufs an. Durch unser `` wird standardmäßig das Bild `HoyerFoto.jpg` angezeigt, in dem Bild werden zudem die beiden Eventhandler `onMouseOver` und `onMouseOut` überschrieben (Zeilen 27 und 28).

Sobald die Maus über dem Foto liegt, wird die Animation über die Methode `runTheShow` gestartet. `runTheShow` greift zunächst auf die global angelegten Variablen `bilder` (ein Array, das die beiden Bildnamen enthält) und `bildindex` (Indexposition in dem Array `bilder`, kann daher nur die Werte 0 oder 1 besitzen) zu.

¹ Ich vermute, die Verwirrung steigt.

```
1  <head>
2  <script language="javascript">
3  <!--
4  var bilder = new Array("zdebel.png", "HoyerFoto.jpg");
5  var bildindex = 1;
6  var ID;
7
8  function runTheShow() {
9
10     bildindex = 1 - bildindex;
11     window.document.images[0].src = bilder[bildindex];
12     ID = window.setTimeout("stopTheShow();runTheShow()", 1000);
13
14 }
15
16 function stopTheShow() {
17
18     window.clearTimeout(ID);
19
20 }
21 //-->
22 </script>
23 </head>
24 <body>
25 ...
26 
30 ...
31 </body>
```

Abbildung 6.1: Es müssen nicht immer animierte GIFs sein. Mit JavaScript können Bildershows häufig flexibler gestaltet werden.

Nachdem das Bild ausgetauscht worden ist (Zeile 11), tritt nun der Zeitgeber in Aktion. Dieser löscht nach Ablauf einer Sekunde (=1000 Millisekunden), den vorangegangenen Zeitauftrag mit der Auftragsnummer ID.²

Im Anschluss daran ruft sich die Methode selbst auf, und der Austausch der Bilder beginnt erneut. Gleichzeitig erhält ID den Wert des aktuellen Zeitgebers (der damit im nächsten Umlauf gelöscht werden kann). Das oben gezeigte Verfahren des Selbstaufrufs wird im Fachjargon als *rekursiv* bezeichnet.

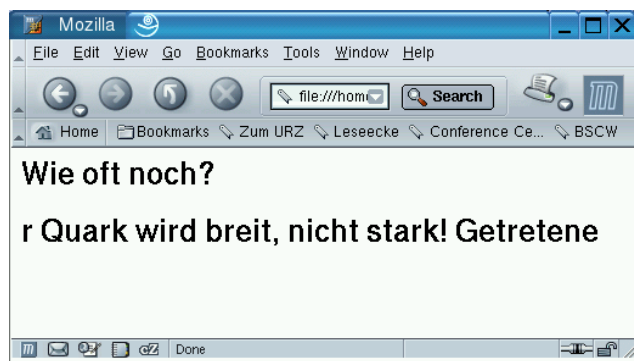
² Beim Erstaufruf der Funktion existiert selbstverständlich *keine Auftragsnummer des Vorgängers*. Dies hat aber auf den weiteren Ablauf keinen Einfluss.

Unterbrochen wird die rekursive Kette schließlich, indem durch die Funktion `stopTheShow` die *aktuelle Auftragsnummer* gelöscht wird.

6.2 Übungsaufgabe

Erstellen Sie ein sogenanntes „Laufband“, das heisst: Sie haben eine HTML-Seite mit einer beliebigen Textzeile (in unserem Fall mit dem Inhalt Getretener Quark wird breit, nicht stark!).

Sobald der Mauszeiger über der Textzeile steht, setzt sich diese in Bewegung indem die Buchstaben nach links rutschen. Zeichen die über den linken Rand hinaus bewegen, werden von rechts kommend wieder eingefügt.



7 Für alle anderen Tage...

Wie nicht anders zu erwarten, hat natürlich auch eine JavaScript-Broschüre irgendwann einmal ein Ende. Sofern Sie den Inhalt aufmerksam gelesen, bzw. dem zugehörigen Seminar aufmerksam gefolgt sind, so haben Sie jetzt jede Menge Werkzeuge um Ihre WWW-Seiten aufzupeppen; darüberhinaus besitzen Sie jedoch auch eine erstklassige Ausgangsposition um sich tiefere Kenntnisse anzueignen.

Dabei stellt sich natürlich immer noch, oder immer wieder, die Frage: Soll JavaScript nun eingesetzt werden (und wenn ja, in welchem Umfang), oder ist es vielleicht doch besser die eigenen WWW-Seiten möglichst scriptfrei zu halten?

Eine klare Antwort auf diese Frage kann und wird es natürlich nie geben. JavaScript *kann* die Funktionalität Ihrer Seite stark *erhöhen*, ein Zuviel *wird* jedoch mit Sicherheit *schaden*. Die folgenden Tipps von Christine Kühnel werden Ihnen, bei Ihren Versuchen das richtige Maß zu finden, bestimmt helfen.

7.1 Nie, Nie, ... Niemals!

Erhalten Sie die Zugänglichkeit

Führen Sie niemanden in die Sackgasse! Benutzen Sie JavaScript niemals um Benutzer auf Ihren Seiten festzuhalten.

Machen Sie wichtige Inhalte immer per HTML zugänglich! Verstecken Sie keine Information in Fenstern, die sich nur mit Hilfe von JavaScript öffnen lassen. Bauen Sie in Ihre Seiten keine Navigationsmöglichkeiten ein, die ohne JavaScript nicht zu benutzen sind.

Denken Sie über serverseitige Lösungen nach! Sofern sich Funktionalitäten, anstelle von JavaScript, mit serverseitigen CGI-Skripten lösen lassen, ziehen Sie diese im Zweifelsfall vor.

Bevormunden Sie nie Ihre Benutzer

Beeinflussen Sie nie die Funktionsmöglichkeiten des WWW-Browsers! Deaktivieren Sie keine üblichen Navigationselemente, oder weisen ihnen andere Funktionen zu. Lösen Sie keine unerwarteten Funktionen aus.

Ändern Sie nie die Größe bereits geöffneter Fenster! Die meisten Benutzer fühlen sich mit der von ihnen gewählten Fenstergröße durchaus wohl, belassen Sie es einfach dabei.

Anhang A Lösungen aller Aufgaben

Aufgaben, Seite 29

```
1  <html>
2  <body>
3  <h2>Glückszahlen</h2>
4  <script language="javascript">
5  <!--
6
7  var vorname = window.prompt("Ihren Vornamen bitte:", "");
8  var gruss = "Guten Morgen ";
9  var laenge = (gruss + vorname).length
10
11  var zahl1 = Math.ceil((49 * Math.random()));
12  var zahl2 = Math.ceil((49 * Math.random()));
13  var zahl3 = Math.ceil((49 * Math.random()));
14  var zahl4 = Math.ceil((49 * Math.random()));
15  var zahl5 = Math.ceil((49 * Math.random()));
16  var zahl6 = Math.ceil((49 * Math.random()));
17
18  window.document.write("<p>" + gruss + vorname + " ,</p>");
19  window.document.write("<p>Deine persönlichen Glückszahlen für heute: " +
20      zahl1 + ", " + zahl2 + ", " + zahl3 + ", " + zahl4 + ", " +
21      zahl5 + " und " + zahl6 + "</p>");
22
23  window.status = "Länge der Textkette: " + laenge;
24
25  //-->
26 </script>
27 </body>
28 </html>
```

Aufgaben, Seite 38

$$ab^2 + cd^2 = abcd$$

```
1  <html>
2  <body>
3
4  <h2>ab<sup>2</sup> + cd<sup>2</sup> = abcd</h2>
5
6  <script>
7  <!--
8
9  var i;
10 var lt;
11 var rt;
12
13 for (i = 1000; i < 10000; i++) {
14
15     lt = Math.floor(i / 100);
16     rt = i % 100;
17
18     if ( (lt*lt + rt*rt) == i ) {
19
20         document.write("<p>" + lt + "<sup>2</sup> + " +
21                         rt + "<sup>2</sup> = " +
22                         i + "</p>");
23
24     }
25
26 }
27
28 //-->
29 </script>
30
31 </body>
32 </html>
```

Würfelstatistik

```
1  <html>
2  <body>
3
4  <h2>Würfelstatistik</h2>
5
6  <script>
7  <!--
8
9  var wurfanz = prompt("Wie häufig soll geworfen werden?", "");
10 var arraysize = 6;
11 var wuerfe = new Array(arraysize);
12 var wurfzahl;
13
14 // Den kompletten Array nullen
15 for (var i = 0; i < wuerfe.length; i++) {
16     wuerfe[i] = 0;
17 }
18
19 for (var i = 1; i <= wurfanz; i++) {
20
21     // Erzeuge eine Zahl mit 0 <= zahl < 6
22     wurfzahl = wuerfe.length * Math.random();
23
24     // Runde diese Zahl ab
25     wurfzahl = Math.floor(wurfzahl);
26
27     wuerfe[wurfzahl]++;
28
29 }
30
31 document.write("<h3>Bei " + wurfanz + " Würfeln entfielen</h3>");
32
33 for (var i = 0; i < wuerfe.length; i++) {
34     document.write("<p> auf die " + eval(i+1) + ": " + wuerfe[i] +
35         " Würfe</p>");
36 }
37
38 //-->
39 </script>
40
41 </body>
42 </html>
```

Aufgabe, Seite 46

```

1  <html>
2
3  <head>
4  <script language="javascript">
5  <!--
6  function gausssum(start, stop) {
7
8  var summe = 0;
9
10 for (var i = start; i <= stop; i++) {
11
12     summe = summe + i;
13
14 }
15
16 return summe;
17 }
18 //-->
19 </script>
20 </head>
21
22 <body>
23 <h2>Das Gausssche Summenmodell</h2>
24 <script language="javascript">
25 <!--
26
27 document.write("<p>Die Summe der Zahlen von 1 bis 100 ist: " +
28                 gausssum(1, 100) + "</p>");
29
30 //-->
31 </script>
32 </body>
33 </html>

```

Anmerkung: Der kleine Gauß hatte sofort erkannt, dass

$$\left. \begin{array}{rcl}
 100 + 1 & = & 101 \\
 99 + 2 & = & 101 \\
 98 + 3 & = & 101 \\
 \dots & & \\
 52 + 49 & = & 101 \\
 51 + 50 & = & 101
 \end{array} \right\} = 50 * 101 = 5050$$

Aufgabe, Seite 65

```
1  <html>
2
3  <head>
4  <script language="javascript">
5  <!--
6  var stationen = new Array('Eins Live', 'Radio Luxemburg', 'Antenne Bayern');
7
8  function getStation() {
9
10     var statsdim = window.document.forms[0].elements['stats'].length
11     var aktelem;
12
13     for (var i = 0; i < statsdim; i++) {
14
15         aktelem = window.document.forms[0].elements['stats'][i];
16
17         if (aktelem.checked) {
18             window.document.forms[0].elements['station'].value = stationen[i];
19         }
20     }
21 }
22
23 }
24
25 function setStation() {
26
27     var statsdim = window.document.forms[0].elements['stats'].length
28     var aktelem;
29
30     for (var i = 0; i < statsdim; i++) {
31
32         aktelem = window.document.forms[0].elements['stats'][i];
33
34         if (aktelem.checked) {
35             stationen[i] = window.document.forms[0].elements['station'].value;
36         }
37     }
38 }
39
40 }
41 //-->
42 </script>
43 </head>
44
45 <body>
46 <h2>Ein Dampfradio</h2>
47
```

```
48 <form>
49 <p><input type="text" size="25" name="station"
50     onBlur="javascript:setStation();"></p>
51 <p><input type="radio" name="stats"
52     onClick="javascript:getStation();">Station 1
53 <input type="radio" name="stats"
54     onClick="javascript:getStation();">Station 2
55 <input type="radio" name="stats"
56     onClick="javascript:getStation();">Station 3
57 </form>
58 </body>
59 </html>
```

Aufgaben, Seite 69

„Cookie“-Aufgabe

```
1  <html>
2
3  <head>
4  <script language="javascript">
5  <!--
6  var stationen = new Array('Eins Live', 'Radio Luxemburg', 'Antenne Bayern');
7
8  function restoreStationOne () {
9
10 var stationcookie = window.document.cookie;
11
12 if ( stationcookie.substr(0,8) == "Station1" ) {
13
14     stationen[0] = stationcookie.substr(9);
15
16 }
17
18 }
19
20 function setStationOne () {
21
22 window.document.cookie = "Station1=" + stationen[0]
23 /* Die folgende Zeile auskommentieren um Station1 dauerhaft zu speichern */
24 ;
25 //             + "; expires=Wednesday, 31-12-03 23:59:59 GMT";
26
27 }
28
29 function getStation() {
30
31     var statsdim = window.document.forms[0].elements['stats'].length
32     var aktelem;
33
34     for (var i = 0; i < statsdim; i++) {
35
36         aktelem = window.document.forms[0].elements['stats'][i];
37
38         if (aktelem.checked) {
39             window.document.forms[0].elements['station'].value = stationen[i];
40         }
41
42     }
43
44 }
45
```

```
46 function setStation() {
47
48     var statsdim = window.document.forms[0].elements['stats'].length
49     var aktelem;
50
51     for (var i = 0; i < statsdim; i++) {
52
53         aktelem = window.document.forms[0].elements['stats'][i];
54
55         if (aktelem.checked) {
56             stationen[i] = window.document.forms[0].elements['station'].value;
57         }
58     }
59 }
60
61 }
62 //-->
63 </script>
64 </head>
65
66 <body onload="javascript:restoreStationOne();"
67     onunload="javascript:setStationOne();">
68
69 <h2>Ein Dampfradio</h2>
70
71 <form>
72 <p><input type="text" size="25" name="station"
73     onBlur="javascript:setStation();"></p>
74 <p><input type="radio" name="stats"
75     onClick="javascript:getStation();">Station 1
76 <input type="radio" name="stats"
77     onClick="javascript:getStation();">Station 2
78 <input type="radio" name="stats"
79     onClick="javascript:getStation();">Station 3
80 </form>
81 </body>
82 </html>
```


Gezoomtes Bild im eigenen Fenster

```
1  <html>
2
3  <head>
4  <script language="javascript">
5  <!--
6  function zoomFenster() {
7
8  var merkmale = "menubar=no, toolbar=no";
9  merkmale = merkmale + ", directories=no, location=no";
10 merkmale = merkmale + ", status=no, scrollbars=yes";
11 merkmale = merkmale + ", width=720, height=420";
12
13 var fenster = window.open("", "", merkmale);
14 var doku = fenster.document;
15
16 doku.write("<html><head><title>Zoom-Fenster</title></head>");
17 doku.write("<body><img name='zoombild' width='100%'></body></html>");
18
19 doku.images['zoombild'].src = window.document.images['feubild'].src;
20
21 }
22 //-->
23 </script>
24 </head>
25
26 <body>
27 <h2>Bildlupe</h2>
28
29 
31 <p>Klicken Sie auf das Bild um eine größere Darstellung zu erhalten.</p>
32
33 </body>
34 </html>
```

Aufgabe, Seite 74

```
1  <html>
2  <head>
3  <script language="javascript">
4  <!--
5
6  var zk;
7  var ID;
8
9  function runTheShow() {
10
11     zk = zk.substr(1) + zk.charAt(0);
12     window.document.getElementById('kette').innerHTML = zk;
13     ID = window.setTimeout("stopTheShow();runTheShow()", 100);
14
15  }
16
17  function stopTheShow() {
18
19     window.clearTimeout(ID);
20
21  }
22  //-->
23 </script>
24 </head>
25 <body>
26   <h2>Wie oft noch?</h2>
27   <script language="javascript">
28     <!--
29     //-->
30   </script>
31   <h2 id='kette'
32     onMouseOver="javascript:zk=window.document.getElementById('kette').innerHTML;
33                   runTheShow();"
34     onMouseOut="javascript:stopTheShow();"
35   >Getreter Quark wird breit, nicht stark! </h2>
36 </body>
37 </html>
```

Anhang I Index

- Arrays, 17
 - Ändern der Größe von, 18
 - Anlegen von, 17
 - arguments, 42
 - Besetzen von, 17
 - length, 18, 42
 - Mehrdimensionale, 18
- Ausgaben, 26
 - Meldungsfenster, 28
 - Statuszeile, 27
- Bezeichner, 14
- case, 33
- Cookies, 66
 - expires, 68
- Datentypen, 15
 - isNaN, 21
 - NaN, 21
 - typeof, 15
- Debugging, 11
- default, 33
- do...while, 37
- DOM, 3, 63
- Eingaben, 26
- Ereignisse, 49
- event, 54
- Eventhandler, 50
 - Überschreiben von, 50
 - Übersicht, 52
 - onBlur, 53
 - onChange, 53
 - onClick, 51, 52, 54
 - onDblClick, 53
 - onError, 52
 - onFocus, 53
 - onKeyPress, 53, 54
 - onLoad, 52
 - onMouseOut, 50, 52
 - onMouseOver, 50, 52
 - onMove, 52
 - onReset, 53
 - onResize, 52
 - onSubmit, 54
 - onUnload, 52
- Events, 49
 - Spezifizieren von, 54
- Fallunterscheidungen, 32
- for, 36
- function, 40
- Funktionen, 14, 39
 - Argumente, 14, 40
 - Variable Argumentlisten, 42
 - für Zeichenketten, 22
 - Mathematische, 22
 - Parameter, 40
 - parseInt, 35
 - Rekursive, 72
- HTML, 1, 2, 24, 49
- id, 63
- if, 32
- Internet Explorer, 7
 - Debugging, 11
- Interpreter, 3, 7, 41
- Java, 3
- JavaScript, 49
 - Console, 11
 - Interpreter, 49
- Kommentar, 9, 19
- Kontrollstrukturen, 31
- Meldungsfenster, 28
- Mozilla, 7
 - Debugging, 11
- name, 63
- NaN, 15, 25
- Netscape, 7
 - Debugging, 11
- Notepad, 7
- Objekt, 59
 - Attribute, 60
 - Methoden, 60

- Opera, 7
- Operationen
 - Arithmetische, 21
 - Modulo, 21
- return, 40
- Schleifen, 35
- Statuszeile, 27
- switch, 33
- Syn, 7
- var, 13
- Variablen, 13, 42
 - Anlegen von, 14
 - Bezeichner, 14
 - Datentypen, 15
 - Boolsche Werte, 15
 - Strings, 16
 - Zahlen, 16
 - Zeichenketten, 16
 - delete, 17
 - Gültigkeitsbereich, 42
 - Globale, 44
 - Löschen von, 17
 - undefined, 14, 17
- Verzweigungen, 32
- while, 35
- window, 27, 45, 55
 - clearTimeout, 71
 - defaultStatus, 28
 - document, 27
 - cookie, 67
 - elements, 60
 - forms, 60
 - getElementById, 63
 - images, 60
 - innerHTML, 63
 - links, 60
 - write, 27
 - history, 68
 - location, 55
 - replace, 55, 68
 - navigator, 66
 - prompt, 26, 28
 - setTimeout, 71
 - status, 27
- Word, 7
- Zeichenketten, 16
 - Aufteilen von, 24
 - Escape-Sequenzen, 16
 - Formatierte, 24
 - Funktionen für, 22
 - big, 24
 - bold, 24
 - charAt, 23
 - concat, 23
 - eval, 25
 - fixed, 24
 - fontcolor, 24
 - fontsize, 24
 - indexOf, 24
 - italics, 24
 - link, 24
 - parseFloat, 25
 - parseInt, 25
 - small, 24
 - split, 24
 - substr, 23
 - substring, 23
 - Konvertieren von Zahlen in, 25
 - leere, 23
 - length, 22
 - Steuerzeichen, 16
 - String(), 25
 - Teilzeichenketten, 23
 - Suchen von, 24
 - Verknüpfen von, 23
 - Zahlen als, 25
 - Zeichen ermitteln in, 23
 - Zeichenanzahl, 22
- Zeitgeber, 71
- Zuweisungen, 20

Literaturverzeichnis

- [1] David Flanagan: JavaScript. Das umfassende Referenzwerk. O'Reilly, 1997.
- [2] Thomas Feuerstack, Jens Vieler: Die kleine HTML-Schule. Fern-Universität, Universitätsrechenzentrum, 1998.¹
- [3] Brian W. Kernighan, Dennis M. Ritchie: Programmieren in C. Hanser, 2. Auflage, 1990.

¹ Kostenlos über die Lesecke des Universitätsrechenzentrums erhältlich:
<http://www.fernuni-hagen.de/urz/lesecke>