JavaScript lernen

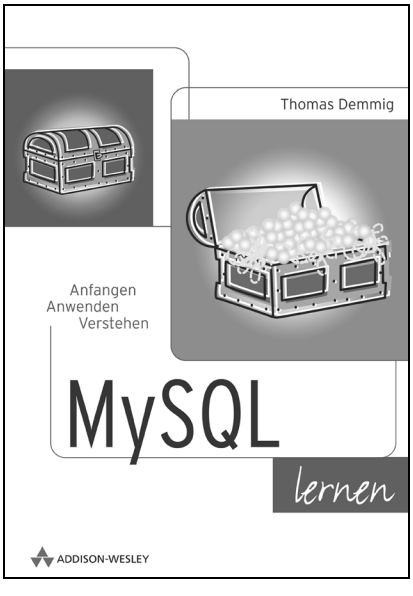
Die Lernen-Reihe

Die Einsteigerreihe. Anfangen, anwenden, verstehen! Ob Sie nun HTML, Java, PHP, Visual Basic, Delphi oder C++ lernen oder mit Visual Basic für Applikationen Ihre Office-Anwendungen optimieren wollen – unsere erfolgreichen Lernen-Bücher verschaffen Ihnen einen schnellen und motivierenden Einstieg. Wichtig ist uns die richtige Mischung aus Theorie und praktischen Übungen.

PHP lernen

Olivia Adler, Hartmut Holzgraefe
280 Seiten, 1 CD-ROM
€ 24,95 [D]/€ 25,70 [A]/sFr 39,50
ISBN 3-8273-2000-3

Mit PHP lassen sich dynamische Webanwendungen effektiv und kostengünstig realisieren. Wer sich bereits mit HTML auskennt und schon erste Webseiten programmiert hat, erhält hier einen praxisorientierten Einstieg in PHP. Am Beispiel eines Content Management Systems lernen Sie, selbst Anwendungen in PHP unter Einbindung von MySQL-Datenbanken zu programmieren.



MySQL lernen

Thomas Demmig
344 Seiten, 1 CD-ROM
€ 24,95 [D]/€ 25,70 [A]/sFr 39,50
ISBN 3-8273-2070-4

MySQL ist der wohl am häufigsten verwendete freie Datenbankserver. Mit diesem Buch erlernen Sie den Umgang mit MySQL und seinen Dienstprogrammen. SQL wird erklärt und an vielen Beispielen nähergebracht. Das Buch erklärt die Grundlagen, erläutert Hintergründe und gibt professionelle Tipps. Eigene Abschnitte zeigen das Zusammenspiel mit PHP und dem Apache Webserver. Übersichtliche Beispiele veranschaulichen die Erläuterungen und Übungsfragen ermöglichen es Ihnen, Ihr Wissen zu überprüfen. Als durchgehende Beispieldatenbank wird die Verwaltung einer Musikschule entwickelt.

PEARSON Education

ADDISON-WESLEY

Daniel Koch

JavaScript lernen

Anfangen, anwenden, verstehen

ADDISON-WESLEY

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Bibliografische Information Der Deutschen Bibliothek

Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über *http://dnb.ddb.de* abrufbar.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.

10 9 8 7 6 5 4 3 2 1

06 05 04 03

ISBN 3-8273-2087-9

Einbandgestaltung:	Barbara Thoben, Köln
Korrektorat:	Alexandra Müller, Oer-Erkenschwick
Lektorat:	Martin Asbach, masbach@pearson.de
Herstellung:	Ulrike Hempel, uhempel@pearson.de
Satz:	mediaService, Siegen
Druck und Verarbeitung:	Media Print, Paderborn

Printed in Germany

Inhaltsverzeichnis

1 Einführung

lernen

Wozu die Tasten sonst noch da sind – ich habe keine Ahnung. Macht auch nichts, denn viele haben nur noch einmal dieselbe Funktion wie andere Tasten oder tun überhaupt nichts. Meine Lieblingstaste dieser Kategorie ist eine, die mit „Pause" beschriftet ist und wirklich nichts tut, wenn man draufdrückt – was die reizvolle philosophische Frage aufwirft, ob sie dann ihre Pflicht tut.

Bill Bryson in „Streiflichter aus Amerika"

1.1 Zielsetzung dieses Buches

Buchaufbau

Das Ziel von „JavaScript lernen" lässt sich einfach umschreiben: Sie als Leser sollen nach der Lektüre dieses Buches anspruchsvolle JavaScript-Programme schreiben können. Das vor Ihnen liegende Buch gliedert sich grundlegend in zwei Teile. In den ersten Kapiteln werden Sie allgemeine Informationen über JavaScript erhalten. Hier wird auf die Syntax, Sprachelemente und Programmiertechniken eingegangen. Im zweiten Teil folgen praktische Anwendungen. Sie werden dort den Umgang mit wichtigen JavaScript-Objekten kennen lernen. Es werden all die Objekte behandelt, die Sie für die Erstellung von Internetseiten benötigen. Angefangen bei der Manipulation des Browserfensters, über die automatische Kontrolle von Formulareingaben, bis hin zu mathematischen Funktionen reichen die vorgestellten Anwendungen. Alle in diesem Buch vorhandenen JavaScript-Programme finden Sie auch auf der beiliegenden CD-ROM. Sie werden also nicht nur die Sprache selbst lernen, sondern erhalten zusätzlich eine JavaScript-Bibliothek mit Anwendungen für die verschiedensten Bereiche.

1.1.1 Was dieses Buch nicht will

ausschließlich JavaScript

Dieses Buch mit einem Umfang von circa 400 Seiten soll Ihnen die Sprache JavaScript beibringen. Dieses Ziel wird mit aller Konsequenz verfolgt.

Die Frage bei der Konzeption dieses Buches war nun, ob Exkurse in andere Sprachen unternommen werden sollen. So böte sich dieses Buch ideal als Plattform für ein HTML-Kapitel an. Dennoch habe ich mich dagegen entschieden. Der Grund hierfür soll Ihnen freilich nicht vorenthalten werden: Ein JavaScript-Buch ist ein JavaScript! Ich sträube mich entschieden gegen Kapitel in einem JavaScript-Buch, die mit solchen Überschriften wie „HTML-Crashkurs“ u.Ä. beginnen. Das Vorhaben, die Vermittlung von HTML-Wissen, mag löblich sein, wurde auf das vor Ihnen liegende Buch dennoch nicht angewandt. Ich kann mir schlicht und ergreifend nicht vorstellen, dass die Käufer dieses Buches nicht grundlegende HTML-Kenntnisse besitzen. Sollte dies dennoch der Fall sein, könnte ein HTML-Kapitel in diesem Buch das fehlende Wissen auch nicht vermitteln. Es gibt weitere Punkte, die nicht in dieses Buch mit einfließen oder nicht ausführlich genug behandelt werden. So fehlen z.B. vollständig die verschiedenen Internetgrundlagen, die Erläuterung existierender Übertragungs-Protokolle sowie ein historischer Ausflug in die Entstehung des Internet. Das mag schade sein, schließlich sind dies alles interessante Aspekte, die gewürdigt werden müssen. Aber auch hier gilt, dass dies den Rahmen des Buches sprengen würde und es für angehende JavaScript-Programmierer auch nicht zwingend notwendige Informationen sind.

Dieses Symbol steht für einen Hinweis, der Sie auf besonders nützliche Informationen aufmerksam machen soll.

Dieses Icon schließlich stellt eine Warnung dar, um Sie auf mögliche Fehlerquellen hinzuweisen.

Besten Dank

Tim Allan für „Hör mal, wer da hämmert“. Der Nationalmannschaft für eine phantastische WM in Japan und Südkorea. Bill Bryson für die witzigsten Reiseberichte aller Zeiten. Nick Hornby für all seine Bücher. Aus der schreibenden Gilde seien zusätzlich Tony Parsons, Bret Easton Ellis und Phillip Roth genannt. Musikalische Weggefährten waren u.a. die Mighty Mighty Bosstones, Frau Doktor und The Offspring.

Selbstverständlich dürfen hier nicht die Menschen fehlen, die mich in den letzten Jahren begleitet und unterstützt haben oder einfach nur da gewesen sind. Stellvertretend seien hierfür Vater (aus dem Piano), Mutter und Bruder genannt. Ein weiterer Gruß geht an den Neu-Studenten, seine Frau, an Erik und den neuen Nachwuchs (Willkommen!). Ein letzter Gruß geht an Rico.

Wieder einmal zuletzt genannt, dennoch nicht so gemeint, gilt mein Dank natürlich auch wieder Sarah. Was soll ich sagen? Besten Dank für

die letzten Jahre, das letzte Silvester (Wahnsinn, oder?) und auch sonst für alles. Wie immer und ausschließlich in holder Absicht (Gedöns) ist dieses Buch Dir gewidmet.

Daniel Koch, Berlin/Ilmenau 2003

1.2 Was ist JavaScript?

Diskussionen um eine Sprache

Und JavaScript ist doch eine Programmiersprache! Diese Aussage lässt sich einfach belegen. Mit JavaScript lassen sich Programme schreiben. Und das ist dann auch schon der Beleg. Unter Programmierern genießt JavaScript dennoch oft einen schlechten Ruf. Dieser ist jedoch nur bedingt begründet. Häufig wird im eigentlichen Sinn nicht JavaScript selbst, sondern werden die erstellten Programme kritisiert. Und hier gebe ich den Kritikern Recht. Viele JavaScript-Programme sind undokumentiert, schlecht strukturiert, mit undeutlichen Bezeichnern versehen und einfach schlecht programmiert. Dies liegt jedoch nicht an der Sprache, sondern am jeweiligen Programmierer. Dieses Buch soll Ihnen Ansätze zeigen, mit denen Sie diese Probleme umgehen und sich einen schlechten Programmierstil gar nicht erst angewöhnen.

eine junge Programmiersprache

JavaScript kann immer noch als junge Programmiersprache bezeichnet werden. Seit 1995 steht mit dieser Sprache HTML-Autoren ein Werkzeug zur Verfügung, mit dem sich Internetseiten optimieren lassen. Trotz vieler für den Einsteiger und Nicht-Programmierer zunächst verwirrender Elemente der Sprachsyntax, wie beispielsweise Schleifen und bedingte Anweisungen, eignet sich JavaScript vorzüglich, um programmieren zu lernen. Erfahrene Programmierer werden diese Elemente kennen, sich aber über manch fehlende Möglichkeit bei der Programmierung wundern. Als Beispiel sei hier die in höheren Programmiersprachen übliche Typdefinition von Variablen genannt.

Viele Anwender unterbinden die Ausführung von JavaScript in ihrem Browser. Dass dies häufig zu Recht geschieht, können Sie bei den Ausflügen auf verschiedene Seiten, die sich JavaScript bedienen, sehen. Denn häufig wird JavaScript als Mittel zur Profilierung für den Seiten-Entwickler eingesetzt. Als krönende Beispiele seien das Deaktivieren der rechten Maustaste, das Nicht-mehr-verlassen-Können der aufgerufenen Seite, das ungefragte Öffnen von neuen Fenstern und die Belegung der Statuszeile mit Text genannt. Durch solche Anwendungen wird die Akzeptanz von JavaScript gemindert. JavaScript sollte stets als Teil von HTML verstanden und zu dessen Unterstützung eingesetzt werden. Anwendungen zum Selbstzweck sollten indes vermieden werden.

1.2.1 Clientseitiges JavaScript

JavaScript-Interpreter

Damit ein WWW-Browser in HTML-Dokumente eingebettete JavaScript-Programme ausführen kann, muss dieser einen JavaScript-Interpreter beinhalten. Es gibt verschiedene Möglichkeiten, um JavaScript auf ein HTML-Dokument anwenden zu können. Der Quellcode kann entweder direkt in das HTML-Dokument eingebettet, er kann jedoch auch in einer separaten Datei abgespeichert werden. Die Funktionsweise von JavaScript stellt sich folgendermaßen dar: Vom Client, also dem WWW-Browser, wird eine HTML-Datei von einem Server angefordert. Dieser sendet den gesamten Quellcode, inklusive HTML- und JavaScript-Bereiche, an den WWW-Browser. Der erhaltene Code wird vom Browser von oben nach unten gelesen und interpretiert.

1.2.2 Serverseitiges JavaScript

Es gibt bessere Möglichkeiten!

Weniger häufig als client- wird serverseitiges JavaScript verwendet. Dies mag daran liegen, dass es geeignetere serverseitige Programmiersprachen, wie beispielsweise PHP, gibt. Auch in diesem Buch wird nicht weiter auf serverseitiges JavaScript eingegangen. Nicht nur würde dies den Umfang dieses Buches sprengen, auch schließe ich mich der Meinung an, dass für den serverseitigen Einsatz andere Sprachen verwendet werden sollten. Das Prinzip clientseitigen JavaScripts soll hier aber dennoch kurz vorgestellt werden: Ebenso wie bei client- wird auch bei serverseitigem JavaScript der Quellcode des Programms in HTML eingebettet bzw. in einer externen Datei gespeichert. Bei serverseitigem JavaScript wird der Quellcode allerdings in Byte-Code übersetzt und in eine ausführbare Datei kompiliert. Sendet der Client nun eine Anfrage an den Server, wird diese Datei durchsucht. Anschließend wird eine HTML-Seite generiert und die entsprechenden Server-Anwendungen werden ausgeführt. All diese Schritte werden von der JavaScript runtime engine übernommen. Als Ergebnis dieser Vorgehensweise werden der aufgerufenen HTML-Datei client- oder serverseitige JavaScript-Elemente hinzugefügt. Die hieraus entstehende neue Seite wird an den WWW-Browser gesandt. Weitere Informationen über serverseitiges JavaScript und die Entwicklungsumgebung Live Wire erhalten Sie auf den Entwicklerseiten von Netscape unter *http://developer.netscape.com/docs/manuals/*.

1.3 JavaScript, JScript und ECMA-Script

Probleme durch fehlende Standards

Das Problem an der Entwicklung von JavaScript-Programmen ist nicht die eigentliche Programmierung. Die Problemanalyse, die bei anderen Programmiersprachen zunächst im Vordergrund steht und sich mit den tatsächlichen Begebenheiten befasst, beschäftigt sich bei JavaScript fast ausschließlich mit der Frage, welcher Browser ein bestimmtes JavaScript interpretieren kann. Grund hierfür ist einzig und allein die Strategie der Browserentwickler, die eigenen Marktanteile zu steigern. Für den Entwickler heißt dies, dass eine simple Anwendung häufig für mehrere Browser unterschiedlich programmiert werden muss.

1.3.1 JavaScript – Chronologie einer Sprache

Die Sprache JavaScript hat im Laufe der Jahre viele Modifikationen erlebt. Anhand des folgenden Abschnitts sollen die Eckpfeiler der JavaScript-Entwicklung kurz beschrieben werden.

1995

Netscape veröffentlicht eine erste Version des Navigators 2.0. Neben solch revolutionären Elementen wie Frames enthält dieser Browser auch die Sprache LiveScript. Hauptentwickler von LiveScript ist Brendan Eich. Der Name LiveScript lehnt sich an die Netscape-Produkte LiveWire, LiveAudio u.Ä. an. Die Sprachsyntax orientiert sich an Java. Aufgrund des Java-Booms wird aus Marketing-Aspekten der Name der Sprache in JavaScript geändert. Es handelt sich hierbei um den Sprachstandard JavaScript 1.0.

1996

Der Netscape Navigator 3.0 wird veröffentlicht. Dieser Browser unterstützt nun JavaScript 1.1. Als wesentliche Neuerungen sind hierbei das so genannte LiveConnect, also die Kommunikation zwischen JavaScript, Java-Applets und Plug-Ins, sowie eine verbesserte Implementierung des Document Object Modells (DOM) zu nennen.

Microsoft veröffentlicht den Internet Explorer 3. Dieser unterstützt zwar JavaScript, offiziell wird hier jedoch von einer Jscript-Unterstützung gesprochen.

ECMA und Netscape reichen JavaScript bei der ISO ein und wollen damit einen Sprachstandard etablieren.

1997

Der Netscape Navigator 4.0 wird veröffentlicht. Unterstützt wird mittlerweile JavaScript 1.2.

Die ECMA veröffentlicht den Sprachstandard ECMA-262. ECMA-Script wird bei der ISO eingereicht.

Microsoft veröffentlich den Internet Explorer 4. Dieser interpretiert JavaScript 1.1. Zusätzlich werden Eigenentwicklungen des Document Object Models eingeführt.

1998

Netscape veröffentlicht den Netscape Navigator 4.5. Dieser Browser unterstützt JavaScript 1.3 und orientiert sich weitestgehend am ECMA-Script-Standard.

ECMA-Script wird von der ISO als Norm verabschiedet.

Netscape stellt JavaScript 1.4 vor.

2000

Netscape veröffentlicht den Navigator 6. Dieser Browser unterstützt JavaScript 1.5.

2002/2003

Derzeit wird JavaScript 2.0 entwickelt. Die Syntax wird sich noch weiter an anderen Programmiersprachen orientieren. Bislang ist jedoch noch keine Spezifikation verabschiedet. Auch eine Interpretation durch einen Browser wird es wohl in absehbarer Zeit nicht geben.

1.3.2 JScript – Der Ansatz von Microsoft

Microsofts Antwort

Obwohl der Internet Explorer JavaScript in weiten Teilen unterstützt, entwickelte Microsoft quasi eine Parallel-Sprache – JScript. Es handelt sich hierbei um eine fast vollständige Implementierung von JavaScript mit einigen Erweiterungen. Und eben diese Erweiterungen machen JScript in vielerlei Hinsicht interessant. Andererseits wird JScript bislang lediglich vom Internet Explorer interpretiert, was einer weiten Verbreitung dieser Sprache im WWW Einhalt gebietet.

erweiterte Möglichkeiten

Obwohl in JScript alle JavaScript-Elemente implementiert sind, liegt hier nicht der große Vorteil dieser Sprache. Denn warum sollte JScript

für die gleichen Objekte eingesetzt werden, die in dieser Form auch in JavaScript zugänglich sind? Viel wichtiger sind solche Erweiterungen, die sich durch JavaScript nicht realisieren lassen. Als Beispiel hierfür sei das File System Object (FSO) genannt. Durch das FSO wird es möglich, auf das Dateisystem sowohl schreibend als auch lesend zuzugreifen. Somit lässt sich mittels einer einfachen Script-Sprache beispielsweise ein Dateimanager programmieren. Ist es doch mit JScript möglich, Dateien zu schreiben, umzubenennen, zu löschen usw. Beachten Sie, dass sich diese Elemente nur auf Windows-Systeme anwenden lassen.

JScript ist sicherlich nicht so mächtig wie Java oder C. Dennoch handelt es sich auch hier um eine Programmiersprache. Das Vorurteil, dass es sich bei JScript um eine reduzierte Programmiersprache handelt, ist nicht richtig. Zwar lassen sich nicht alle Anwendungen über JScript realisieren, dennoch erfüllt diese Sprache die Anforderungen, die an eine moderne Script-Sprache gestellt werden. Trotz einiger Unterschiede zu JavaScript ist die Notation und Syntax beider Sprachen grundlegend identisch. Es kann also festgehalten werden, dass die Beherrschung von JavaScript unweigerlich auch zu der Beherrschung von JScript führt. Ausgenommen sind hierbei die JScript-eigenen Objekte und Methoden. Nähere Informationen zu JScript finden Sie auf den Entwicklerseiten von Microsoft unter *http://www.microsoft.com/JScript/us/Jstutor/Jstutor.htm*.

1.3.3 ECMA – Ein Standard für alle

ISO und die Standardisierung

Die 1961 gegründete Europe-based ECMA organization wurde 1994 in ECMA - European association for standardizing information and communication systems - umbenannt. Sie widmet sich der Entwicklung von Standards in den Bereichen der Informations- und Kommunikationssysteme. Die ECMA versucht seit 1996, einen Standard für JavaScript durchzusetzen. 1997 reichte die ECMA, in Zusammenarbeit mit Netscape, den Sprachstandard ECMA-262 bei der International Organization for Standardization (ISO) ein. 1998 wurde ECMA-Script von der ISO als Norm ISO/IEC 16262 verabschiedet. Der Kampf um einen einheitlichen Standard in Sachen Web-Scriptsprache ist aber bis heute nicht gewonnen. Zwar beteuern Netscape und Microsoft permanent eine Berücksichtigung des ECMA-Script-Standards, die Realität sieht jedoch anders aus. Es wird sicherlich versucht, die ECMA-Spezifikation einzuhalten, dennoch ist dies bis jetzt nicht vollständig gelungen. Gleichwohl bleibt zu sagen, dass sich durchaus Verbesserungen im Hinblick auf Sprachstandards ergeben haben. Als Beispiel sei hier die Entscheidung von Netscape genannt, in der Navigator-Version 6 das Prinzip von Layern für Dynamisches HTML nicht mehr aufrechtzuerhalten. Weitere Informationen zur ECMA und zu ECMA-Script erhalten Sie unter *http://www.ecma.ch*.

1.3.4 Vor- und Nachteile

ein ausgewogenes Verhältnis

Wie jede andere Programmiersprache bleibt es auch im Umgang mit JavaScript nicht aus, dass diese Sprache neben den Vorteilen auch Nachteile hat. Die Nachteile beziehen sich vor allem auf vorhandene Sicherheitslücken, die im Umgang mit JavaScript ständige Begleiter sind. Obwohl auf die Sicherheitskriterien im Laufe dieses Buches noch häufiger eingegangen wird, sollen bereits an dieser frühen Stelle grundlegende Probleme genannt werden. Bei aller Euphorie, die Entwickler von JavaScript-basierten Anwendungen empfinden mögen, sei an dieser Stelle ausdrücklich darauf hingewiesen, dass JavaScript längst nicht ein solch mächtiges Werkzeug darstellt, wie dies von den Erfindern dieser Sprache allzu oft proklamiert wird.

1.3.5 Vorteile von JavaScript

ein guter Einstieg in die Programmierung

Unser erster Blick soll den Vorteilen von JavaScript gelten. Als größte Vorteile dieser Sprache sind sicherlich ihre leichte Syntax und die hieraus resultierende schnelle Erlernbarkeit zu nennen. Im Gegensatz zu höheren Programmiersprachen wie beispielsweise C oder C++ bleiben im Umgang mit JavaScript komplexe Anwendungen zumeist aus. Auch dies ist freilich ein Grund für die völlig neu entstandene Klientel von Programmierern, die sich hinsichtlich JavaScript häufig aus Designern, Homepage-Gestaltern usw. zusammensetzt. Als weiterer wichtiger Punkt, und dies ist wohl auch ein Grund für die rasante Verbreitung von JavaScript, ist der geringe bis gar nicht vorhandene finanzielle Aufwand zu nennen. So benötigt man, und auch hier sollen als Vergleich wieder andere Programmiersprachen herhalten, für die JavaScript-Entwicklung keine kostenintensiven Entwicklungsumgebungen, sondern im einfachsten Fall einen Texteditor und einen JavaScript-fähigen Browser. Da JavaScript für den Einsatz im World Wide Web konzipiert wurde, erscheint es wenig verwunderlich, dass diese Sprache in dieser Hinsicht ihre wahren Stärken zeigen kann. Besonders großer Beliebtheit erfreuen sich Seiten, die eine gewisse Art an Interaktion mit dem Anwender zulassen. So eignet sich JavaScript beispielsweise vorzüglich, um direkt auf Nutzereingaben reagieren zu können. (Exemplarisch sei hier die Überprüfung von Benutzereingaben innerhalb von Formularen genannt.) Um das enorme Potenzial von JavaScript zu verdeutlichen, werden nachfolgend einige ausgewählte Anwendungsmöglichkeiten dieser Sprache vorgestellt. Bereits diese Auswahl macht deutlich, warum JavaScript in den Augen vieler Webdesigner nach wie vor das Nonplusultra hinsichtlich der einzusetzenden Sprachen des World Wide Web ist.

Formularvalidierung

Wie bereits kurz umrissen, lässt sich JavaScript vorzüglich für die Überprüfung von Benutzereingaben innerhalb von Formularen verwenden. So kann beispielsweise durch ein geeignetes Script eine E-Mail-Adresse auf ihre Gültigkeit hin überprüft werden. Gleiches gilt natürlich auch für Telefon- und Faxnummern, für Postleitzahlen und alle weiteren Nutzereingaben.

Browserweichen

Eine der größten Schwächen des Internet, zumindest aus Programmierersicht, ist die unterschiedliche Behandlung von Internetseiten in den jeweiligen Browsern. Nicht genug, dass sich die Konkurrenzprodukte bislang nicht auf einen einheitlichen Standard einigen konnten, auch zwischen den einzelnen Browserversionen der jeweiligen Hersteller bestehen zum Teil gravierende Unterschiede in der Behandlung von JavaScript-Quellcode. Mögen diese Interpretationsunterschiede den JavaScript-Entwickler auch vor zahlreiche Probleme stellen, eines ist jedoch gewiss: JavaScript bietet, und dies mag paradox klingen, hierfür die geeignete Lösung – das `navigator`-Objekt. Durch dieses lässt sich sowohl der vom Anwender verwendete Browser wie auch dessen Version auslesen. Somit steht dem Entwickler eine Möglichkeit zur Verfügung, mit deren Hilfe sich Internetseiten exakt für den vom Anwender verwendeten Browser programmieren lassen.

Kombinierbar

JavaScript bietet zwar für viele Anwendungen die richtige Lösung, aber auch mit dieser Sprache wird man schnell an seine Grenzen stoßen. Beispielhaft hierfür sei ein Datenbankzugriff genannt. Allein dieser Datenbankzugriff, ohne den nur noch wenige professionelle Internetauftritte auskommen, stellt JavaScript-Entwickler vor Probleme. Für solche Anwendungen sollten besser geeignete Sprachen wie Perl, Cold Fusion oder PHP verwendet werden. Da diese Sprachen aber wiederum für clientseitige Anwendungen nur bedingt geeignet sind, bietet sich eine Verschmelzung dieser Sprachen mit JavaScript an. So ließe sich beispielsweise der Datenbankzugriff per Cold Fusion realisieren und die erhaltenen Informationen würden per JavaScript clientseitig weiterverarbeitet werden.

Die genannten Einsatzfelder von JavaScript sollen an dieser Stelle nur einen ersten Einblick über das Potenzial von JavaScript bieten. Im Laufe dieses Buches werden weitere vorgestellt und anhand von zahlreichen Beispielen praxisnah veranschaulicht. Aber bereits die hier gezeigte Auflistung offenbart, dass sich JavaScript für die Realisierung komplexer Anwendungen eignet.

1.3.6 Nachteile von JavaScript

Sicherheitsmängel und Systemabstürze

JavaScript ist leicht zu erlernen und vor allem für den Einsatz im World Wide Web konzipiert worden. Besonders der erste der hier genannten Punkte, die leichte Erlernbarkeit, ist eines der Probleme von JavaScript. Zwar lassen sich mit geringstem Aufwand einfache Anwendungen erstellen, bei komplexen Problemen stoßen JavaScript-Entwickler aber schnell an ihre bzw. die Grenzen dieser Sprache. Die folgende Auflistung beschreibt, welche Nachteile der Umgang mit JavaScript mit sich bringt. Problemlösungen können an dieser Stelle jedoch nicht geboten werden, da es sich hierbei um nicht veränderbare Tatsachen handelt, mit denen sich jeder Entwickler gleichermaßen konfrontiert sieht. Eine Auswahl der bekanntesten und gravierendsten Fehler bzw. Nachteile zeigen die folgenden Abschnitte.

Gravierende Sicherheitsrisiken

Obwohl von Netscape einige Sicherheitsmodelle wie das „Data Tainting" bereits sehr früh in JavaScript integriert wurden, schließen diese Modelle nicht 100%ig die oft verheerenden Sicherheitslücken. Zwar ist JavaScript recht zuverlässig, was simple Anwendungen betrifft, dennoch treten immer wieder schwere Sicherheitsmängel auf, die ein sicheres Arbeiten nicht mehr zulassen. Exemplarisch hierfür soll die folgende Syntax gelten.

```
document.write('<APP');
document.write('LET');
document.write('name="Einbruch" code="Angriff.class">');
document.write('</APP') ;
document.write('LET>') ;
```

Ohne tiefer in die Syntax von JavaScript eintauchen zu wollen, zeigt dieses Beispiel, mit wie wenig Aufwand sich so genannte Tag-Filter von Firewalls umgehen lassen. Das Prinzip ist denkbar einfach. Angenommen, der erwähnte Tag-Filter dient dem Aussortieren aller Anwendungen, in denen das **<APPLET>**-Tag vorkommt. Nutzt man nun die Möglichkeit des dynamischen Erzeugens von HTML-Code via **write()**-Methode, kann die Filterung umgangen werden. Diese Methode erlaubt es, den anzuwendenden HTML-Quellcode, in diesem Fall das **<APPLET>**-Tag, erst auf dem Client zusammenzusetzen. Für das zuvor aufgeführte Beispiel hätte das die Ausführung des Java-Applets `Einbruch` zur Folge.

Leistungsminderung und Systemabstürze

Endlosschleifen

JavaScript eignet sich hervorragend, um die Systemleistung eines Rechners gen null zu bringen oder diesen gänzlich abstürzen zu lassen. Eine beliebte Methode, deren Anwendung ebenso simpel wie effizient ist, stellt das permanente Öffnen neuer Browserfenster dar. Werden ausreichend neue Fenster geöffnet, kann selbst das effizienteste System an den Rand seiner Leistungsfähigkeit getrieben werden. Eine weitere Möglichkeit, um Systeme zum Absturz zu bringen, sind die in JavaScript leicht zu realisierenden Endlosschleifen. Hiervon müssen aber nicht nur Besucher von Internetseiten betroffen sein. Auch JavaScript-Entwickler sehen sich in einigen Fällen diesem Problem ausgeliefert. Schon bei geringfügig fehlerhaftem Quellcode kann es zu einer solchen Endlosschleife kommen. Als weitere Fehlerquellen bezüglich der Leistungsminderung sind ständig generierte `alert()`-Fenster und das Blockieren der rechten Maustaste zu nennen.

Scripts sind einsehbar

Die Erstellung von komplexen JavaScripts kann enorme Zeit in Anspruch nehmen. Ebenfalls nicht zu vernachlässigen sind die Kreativität und Leistungsbereitschaft, die viele Entwickler aufbieten, um anspruchsvolle Anwendungen zu erstellen. Viele JavaScript-Programmierer nehmen diese Mühen gerne auf sich. Umso ärgerlicher ist es da, wenn die erstellten Scripts innerhalb fremder Anwendungen verwendet werden und dies zudem ohne Einverständnis des Autors geschieht. Zwar etabliert sich eine neue Klientel von „Quellcode-Bedienern“, und zwar solchen, die einen bestehenden Copyright-Hinweis stehen lassen und somit noch die wahre Quelle des Scripts bekannt geben, aber diese Gruppe ist bislang die Ausnahme. Sicherlich, das World Wide Web ist ein Platz, in dem offen gelegtes Wissen und Transparenz vorhanden sein sollten. Aber auch diese sehr lobenswerten Ambitionen sollten nicht als rechtsfreier Raum begriffen werden. Beachten Sie, dass auch im Internet das Urheberrecht gilt und Sie in diesem Sinn, auch aufgrund der Fairness gegenüber dem JavaScript-Entwickler, handeln sollten. Abschließend bleibt zu sagen, dass jedes JavaScript vom Anwender einsehbar ist. Hier helfen weder die Auslagerung des Quellcodes in eine externe Datei noch das Blockieren der rechten Maustaste oder die Verwendung von Frames. Hoffen Sie also auf die Ehrlichkeit Ihrer Mitbürger.

JavaScript ist nicht immer ehrlich

Wer kennt das nicht? Ein Blick auf die Statuszeile in Kombination mit dem Berühren eines Hyperlinks mit der Maus genügt, um den Ziel-URL zu kennen. Falsch! Denn durch den Einsatz von JavaScript kann die Sta-

tuszeile so manipuliert werden, dass hier nicht der richtige URL, sondern anderweitige Informationen bzw. ein gänzlich anderes Verweisziel angezeigt werden. Besonders fatal ist dieses Vorgehen vor allem dann, wenn sicherheitsrelevante Dateien in diese Manipulation mit einbezogen werden. Grundsätzlich gilt, obwohl auch hier die Ausnahme die Regel bestätigt, die Statuszeile sollte tabu sein. Dies gilt vor allem im Hinblick auf notwendige Informationen, wie beispielsweise die Ladezeit der aufgerufenen Seite, auf die viele Anwender nur ungern wegen eines Lauftextes verzichten möchten.

1.3.7 Fazit

dennoch für gut befunden

JavaScript ist keineswegs eine sichere Programmiersprache. Dennoch reichen die Sicherheitsmechanismen, die mit der Weiterentwicklung dieser Sprache zahlreicher werden, aus, um die notwendigsten Vorkehrungen bieten zu können. Dessen ungeachtet sei an dieser Stelle, und das mag ungewöhnlich für ein JavaScript-Buch sein, darauf hingewiesen, dass die Implementierung von JavaScript innerhalb des Browsers deaktiviert werden kann. Mag dies auch die effizienteste Methode zum Umgehen von Sicherheitsrisiken darstellen, lassen sich bei deaktiviertem JavaScript zahlreiche Seiten nur unzureichend bzw. gar nicht betrachten. Hier ist also jeder Anwender gefordert, zwischen dem Für und Wider von JavaScript abzuwägen.

1.3.8 Was kann JavaScript

komplexe Anwendungen inklusive

Die Einsatzgebiete von JavaScript sind ebenso vielfältig wie die Vorurteile gegenüber dieser Sprache. Dennoch lassen sich neben einfachen Programmen auch komplexe Projekte realisieren. Im WWW hat sich JavaScript zur vorherrschenden Sprache für die Gestaltung dynamischer Webseiten herauskristallisiert. Im Folgenden soll aufgezeigt werden, welche Einsatzgebiete für JavaScript geradezu prädestiniert sind.

Interaktion mit dem Dokumentinhalt

Über das `document`-Objekt und dessen Objekte wird die Interaktion mit dem WWW-Dokument möglich. So lassen sich beispielsweise durch das `images`-Objekt Bilder so ansprechen, dass diese manipuliert und Grafikwechsel realisiert werden können. Neben Grafikwechseln ist die Validierung von Formulareingaben wohl das vorherrschende Einsatzgebiet von JavaScript. Diese Form der Kontrolle von Nutzereingaben lag lange Zeit in dem Aufgabenbereich von CGI-Scripten. Der Nachteil hiervon ist allerdings, dass diese Vorgehensweise sehr „netzlastig“ ist. Die eingegebe-

nen Daten müssen zunächst an den Server gesandt, dort überprüft und die Ergebnisse wieder zurück an den Client geschickt werden. Formularvalidierungen auf JavaScript-Basis erfolgen hingegen, so es sich denn um Client Side JavaScript handelt, auf der Client-Seite. Somit können Eingaben auch ohne Netzverbindungen überprüft und entsprechende Meldungen ohne zusätzliche Netzlast dem Nutzer angezeigt werden.

Interaktion mit dem Nutzer

Event-Handling

Das mächtige Mittel des Event-Handlings bietet Möglichkeiten, um auf Nutzereingaben reagieren zu können. Zu diesem Zweck stehen in JavaScript zahlreiche Event-Handler zur Verfügung. So kann beispielsweise durch den Event-Handler `onmouseover` eine Funktion dann ausgeführt werden, wenn der Anwender mit der Maus über einen definierten Bereich des Dokuments fährt. Das Event-Handling kann aber ebenso dafür verwendet werden, um Fehler zu unterdrücken, Formulare abzusenden und auf Tastatureingaben zu reagieren.

Browserkontrolle

JavaScript eignet sich vorzüglich zur Manipulation von WWW-Browsern. So lassen sich beispielsweise mittels der `history`- und `location`-Objekte die Vor- und Zurück-Button des Browsers in Dokumenten simulieren. Das `window`-Objekt hingegen ermöglicht u.a. die Darstellung eigener Texte in der Statusleiste des Browsers. Ein weiteres Beispiel für eine Browsermanipulation ist das Öffnen neuer Browserfenster. Hierbei können neben der Größe des neuen Fensters auch dessen Position und Inhalt exakt definiert werden. Andere Möglichkeiten für die Manipulation des WWW-Browsers sind die Darstellung von Dialogboxen sowie das automatische Schließen von Fenstern.

WWW-Dokumente kontrollieren

Dokumente manipulieren

Mit JavaScript lassen sich alle Elemente einer WWW-Seite manipulieren. Zu diesem Zweck wird das `document`-Objekt eingesetzt. Anhand dessen lassen sich beispielsweise Text und Hintergrundfarbe gestalten. Ebenso ist der dynamische Austausch von Grafiken möglich. Wichtiger als diese Einzelanwendungen ist jedoch die Betrachtung des Ganzen. So können all die genannten, plus zahllose weitere, in Abhängigkeit vom verwendeten Browsertyp angewandt werden. Einem Nutzer mit dem Internet Explorer 3 kann somit eine andere Seite angezeigt werden als einem Anwender mit dem Netscape Navigator 6. Mittels JavaScript kann der gesamte Inhalt einer WWW-Seite dynamisch generiert werden. Hierbei kommt die Möglichkeit von JavaScript zur Anwendung, dass

HTML-Code erzeugt werden kann. Wenn es die Anwendung erfordert, können komplette Seiten per JavaScript in Abhängigkeit vom verwendeten Browser generiert werden.

Protokollierung von Client-Zuständen

Personalisierung von Internetseiten

JavaScript ermöglicht das Speichern von Dateien auf dem Client-Rechner. Dass es sich hierbei um keine gefährlichen Daten handeln kann, dafür sorgt ein Schutzmechanismus, der nur das Speichern von Daten gestattet, die von sehr geringem Umfang sind. Die hier eingesetzte Technologie wurde ebenso wie JavaScript von Netscape entwickelt und lautet Cookies. Solche Cookies werden temporär oder permanent auf dem Client-Rechner gespeichert und ermöglichen, das Nutzerverhalten des Anwenders „auszuspionieren". Genutzt werden Cookies vor allem für die Personalisierung von Webseiten. So lassen sich für einen Anwender exakt auf dessen Bedürfnisse abgestimmte Seiten gestalten. Hierzu wird beim ersten Besuch des Anwenders auf einer Seite ein Cookie auf dessen System gespeichert. Wird die Seite das nächste Mal aufgerufen, werden die gespeicherten Informationen ausgelesen und für den jeweiligen Verwendungszweck genutzt. Es gibt sicherlich zahlreiche Stimmen gegen einen Cookie-Einsatz. Ich möchte in diesem Zusammenhang nur darauf hinweisen, dass jeder Anwender die Möglichkeit hat, das speichern von Cookies zu unterbinden. Eine meinen Anforderungen entsprechende Internetseite ist mir aber das „Risiko" von Cookies allemal wert.

Weitere Einsatzmöglichkeiten

JavaScript kann freilich zu weit mehr Anwendungen genutzt werden, als bisher deutlich geworden ist. Aus diesem Grund richtet sich die folgende Aufzählung demnach auch mehr nach elementaren Seitenelementen, die in dieser Form durch HTML nicht kreiert werden können.

- Das aktuelle Datum und die Uhrzeit können generiert werden.
- Das Überprüfen von Anwendereingaben in Formularfeldern ist möglich.
- Der Zugriff auf Java-Applets kann realisiert werden.
- Meldungsfenster können angezeigt werden.
- Es kann auf Anwenderereignisse reagiert werden.
- Anwenderdaten können abgefragt werden.

Die Liste ließe sich beliebig fortsetzen. An dieser Stelle soll sie jedoch ausschließlich dazu dienen, einen ersten Einblick in die vielfältigen Einsatzmöglichkeiten von JavaScript zu bieten.

1.3.9 Was kann JavaScript nicht

keine Wunderwaffe

JavaScript wird im Normalfall auf dem Client ausgeführt. Diese Tatsache trägt in gewissem Maß zu dem Misstrauen gegenüber JavaScript bei. Denn schließlich ist der Autor des Programms, welches auf dem eigenen Rechner ausgeführt wird, zumeist nicht bekannt. Wer lässt denn gerne Programme auf seinem System laufen, deren Herkunft und Zweck nicht zu ermitteln sind? Aber eben in dieser vermeintlichen Schwäche liegt auch eine Stärke von JavaScript. Während beispielsweise in Pascal Daten auf dem Rechner des Anwenders ohne Nachfrage gelöscht werden können, ist etwas Vergleichbares per JavaScript nicht möglich. Überhaupt gilt, dass durch JavaScript-Programme dem Anwender nicht geschadet werden kann. Selbstverständlich existieren von dieser Regel Ausnahmen, die jedoch zumeist nicht an dem Sprachkonzept liegen. Vielmehr handelt es sich hierbei um Browser-Bugs, die zumeist schnell erkannt und durch ein Patch behoben werden. Die folgenden Punkte führen einige Elemente auf, die JavaScript zwar häufig nachgesagt werden, denen diese Sprache jedoch nicht gerecht wird.

- Es können keine E-Mails automatisch und somit ohne Wissen des Anwenders verschickt werden.
- JavaScript kann keine Grafiken erstellen. Es ist lediglich möglich, Grafiken, wie eben auch in HTML, anzuzeigen.
- Es können keine Daten auf dem Rechner des Anwenders gespeichert werden. Als Ausnahme hiervon sind Cookies zu betrachten.
- Durch JavaScript können keine fremden Fenster geschlossen werden.
- JavaScript gestattet das Generieren von rahmenlosen Fenstern nicht.

Auch diese Liste ist selbstverständlich nicht vollständig und auch nicht in jedem Fall zutreffend. Zwar lassen sich durch reines clientseitiges JavaScript keine Dateien auf dem Rechner speichern, durch das File System Objekt und die Microsoft-Sprache JScript ist dies jedoch sehr wohl möglich. Die gleiche Einschränkung gilt auch für Grafiken. Auch diese lassen sich durch das Anwenden von Filtern und JavaScript dynamisch manipulieren. Aber auch hier gilt, dass es sich um eine reine Microsoft-Entwicklung handelt, die lediglich im Internet Explorer verwendet werden kann.

1.4 Das erste Script – Hello World

der erste Kontakt

Den Einstieg in die Programmierung mit JavaScript soll „Hello World“ liefern. Ein solches Programm wird standardgemäß in allen Programmiersprachen dazu verwendet, um einen Einstieg in die neue Sprache zu ermöglichen. Sie als angehender JavaScript-Programmierer werden durch dieses Script Ihr erstes Programm schreiben und das Ergebnis im Browser ansehen können. Das gezeigte Beispiel dient dazu, den Text `Hello World` im Browser als normalen Text darzustellen. Die folgende Syntax zeigt den gesamten Programmcode, der zur Ausführung der genannten Aufgabe benötigt wird. Wie Sie sehen, handelt es sich hierbei um eine Kombination aus HTML- und JavaScript-Elementen. Wobei einige HTML-Elemente die Formatierung der Ausgabe übernehmen. Hierdurch wird der Text `Hello World` zentriert und in einer Schriftgröße von **+3** dargestellt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<DIV align="center">
<SCRIPT type="text/JavaScript">
<!--
  document.write("<FONT size=+3>");
  document.write("Hello World!");
  document.write("</FONT>");
//-->
</SCRIPT>
</DIV>
</BODY>
</HTML>
```

Listing 1.1: Die Ausgabe von `Hello World`.

Öffnen Sie einen Texteditor Ihrer Wahl und geben Sie hier den Quelltext ein. Abbildung 1.1 zeigt den Programmcode im Windows-Texteditor. Diesen finden Sie unter START PROGRAMME ZUBEHÖR EDITOR.

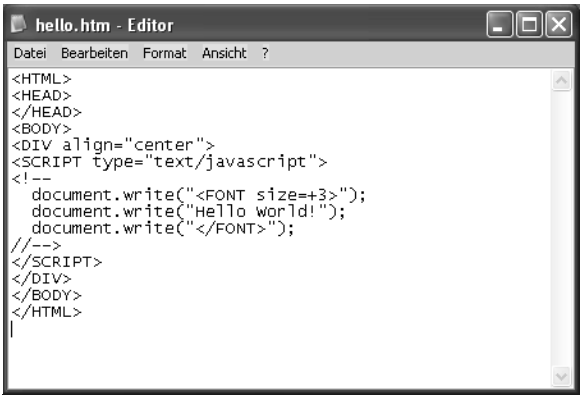


Abbildung 1.1: Der Programmcode im Windows-Editor

Speichern Sie diese Datei unter *hello.htm* ab. Beachten Sie, dass Sie bei dem Dateityp im Dialogfenster SPEICHERN UNTER alle Dateitypen auswählen. Rufen Sie anschließend die abgespeicherte Datei in Ihrem WWW-Browser auf. Es müsste sich Ihnen nun ein ähnlicher Anblick wie in Abbildung 1.2 bieten.

Abbildung 1.2: Hello World im Netscape Navigator 7.0

Der erste Schritt in Richtung JavaScript-Programmierung ist somit erfolgreich getan. Sie wissen nun, wie sich JavaScript schreiben und abspeichern lässt. Ebenso haben Sie einen ersten Eindruck davon gewonnen, wie WWW-Browser JavaScript-Programme interpretieren und darstellen.

2 Grundlagen

lernen

Ziele dieses Kapitels

In diesem Kapitel werden Sie die Grundlagen von JavaScript kennen lernen. Es geht hier jedoch weniger um die eigentliche Entwicklung von JavaScript-Programmen. Vielmehr wird vermittelt, welche Voraussetzungen erfüllt sein sollten, um in die Welt der JavaScript-Programmierung eintauchen zu können. So werden hier die benötigten Werkzeuge wie Editoren, Browser und Script-Debugger vorgestellt. Zudem erfahren Sie, wie JavaScript in HTML-Dateien eingebunden bzw. ausgelagert werden kann.

2.1 Voraussetzungen

Grundsätzlich benötigen Sie für die Erstellung von JavaScript-Programmen nichts weiter als einen Texteditor. Für die Kontrolle des geschriebenen Codes wird der Einsatz eines Browsers empfohlen. An dieser Stelle wird nicht auf das Einbinden von JavaScript-Programmen in HTML-Dokumente eingegangen. Vielmehr sollen hier die Werkzeuge vorgestellt werden, die zum Erstellen von JavaScript-Code notwendig (oder auch nicht) sind.

2.1.1 Editoren

Notepad, Ultra Edit und Co.

Mittlerweile existieren für die Erstellung von JavaScript-Programmen zahlreiche Editoren. Deren Qualität lässt in vielerlei Hinsicht allerdings zu wünschen übrig. Häufig handelt es sich hierbei lediglich um Texteditoren, die eine integrierte JavaScript-Objektreferenz ihr Eigen nennen. Demzufolge ist die Anschaffung eines reinen JavaScript-Editors wohl zu überlegen. Nicht nur, dass sich die Kosten bei professionellen Produkten nur schwerlich rechtfertigen lassen, schwerer wiegt, dass es Alternativen gibt. Für Windows-Anwender sei für die Erstellung der ersten Programme der Notepad-Editor genannt. Bei diesem Programm handelt es sich um einen reinen ASCII-Editor, mit dem sich JavaScript-Code (und anderes) leicht erstellen lässt. Für mehr Komfort eignen sich solche Edi-

toren, die Elemente wie Syntaxhervorhebung und ein komfortables Einrücken ermöglichen. Exemplarisch für diese Gattung von Editoren sei hier Ultra Edtit genannt. Egal für welchen Editor Sie sich entscheiden, für JavaScript gilt Folgendes: JavaScript-Programme werden unter der Dateierweiterung .js abgespeichert. Hiervon ausgenommen sind alle die Programme, die direkt innerhalb einer HTML-Datei geschrieben werden. Mehr über das Einbinden von JavaScript-Programmen erfahren Sie innerhalb des Abschnitts 2.2. Selbstverständlich ist die Wahl eines Editors reine Geschmackssache. Einschlägige Fachzeitschriften bieten Editoren auf den beigefügten CD-ROMs an. Hier kann jeder ambitionierte Entwickler selbst testen, welches Programm das für ihn geeignetste ist. Um Ihnen einen ersten Eindruck vom Aussehen eines reinen JavaScript-Editors zu verschaffen, zeigt die folgende Abbildung den JS Styler in der Version 3.0. Dieser Editor kann unter *http://www.maxro.de* als Demoversion heruntergeladen werden.

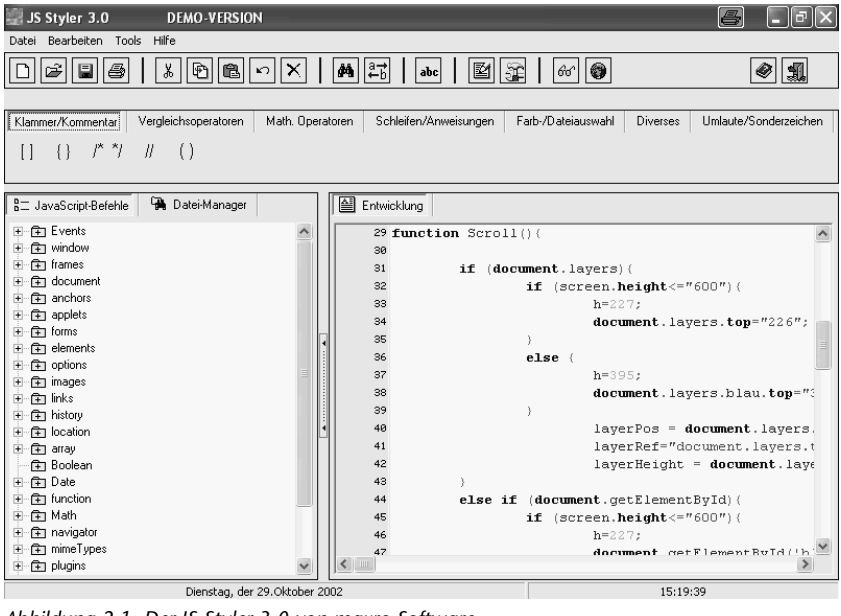


Abbildung 2.1: Der JS Styler 3.0 von maxro Software

Es wird auf den ersten Blick deutlich, dass trotz eines JavaScript-Editors Programmierkenntnisse vorhanden sein müssen. Ohne diese lassen sich keine lauffähigen JavaScript-Programme erstellen. Dies steht im krassen Gegensatz zu WYSIWYG-Editoren zur Erstellung von HTML-Dateien. Viele dieser Editoren sind mittlerweile so ausgereift, dass für die Erstellung von HTML-Seiten keinerlei Kenntnisse in der HyperText Markup Language mehr vorhanden sein müssen. Halten wir also fest, dass JavaScript-Editoren keine fertigen Programme schreiben. Ausnahmen hier-

von bilden einzelne JavaScript-Programme, die häufig in Editoren integriert sind. Als Beispiel hierfür sei Dreamweaver von Macromedia genannt. Zwar ist dies kein JavaScript-Editor, dennoch lassen sich hier ohne Programmierkenntnisse Grafikwechsel in JavaScript realisieren.

2.1.2 Browser

Unterschiedliche Browser sind ein Muss.

Ich werde mich an dieser Stelle nicht an dem „Kampf der Browser" beteiligen. Zum einen werde ich von keinem dieser Konzerne bezahlt und zum anderen haben Sie sicherlich bereits Ihren Lieblingsbrowser gefunden. Vielmehr sei an dieser Stelle darauf hingewiesen, dass JavaScript-Programmierer in jedem Fall mehrere Browser auf ihrem System installiert haben sollten. Denn zu groß sind die Unterschiede in der Umsetzung von JavaScript, als dass man seine Programme lediglich in einem Browser testen kann. Als Faustregel sollte gelten, dass in jedem Fall alle JavaScript-Programme mit dem Internet Explorer und dem Netscape Navigator getestet werden sollten. Zusätzlich schadet es nicht, dass auch andere Browser in die Testphase mit einbezogen werden. Als ein solcher anderer Browser sei hier Opera genannt. Ob nun JavaScript-Code tatsächlich für alle Eventualitäten gewappnet sein muss, ist sicherlich eine Streitfrage. Diese kann jedoch ausschließlich von Fall zu Fall entschieden werden. Als Kriterien gelten hier die Anforderungen des Auftraggebers, die Vorgaben des eigenen Unternehmens sowie die Größe des Projekts.

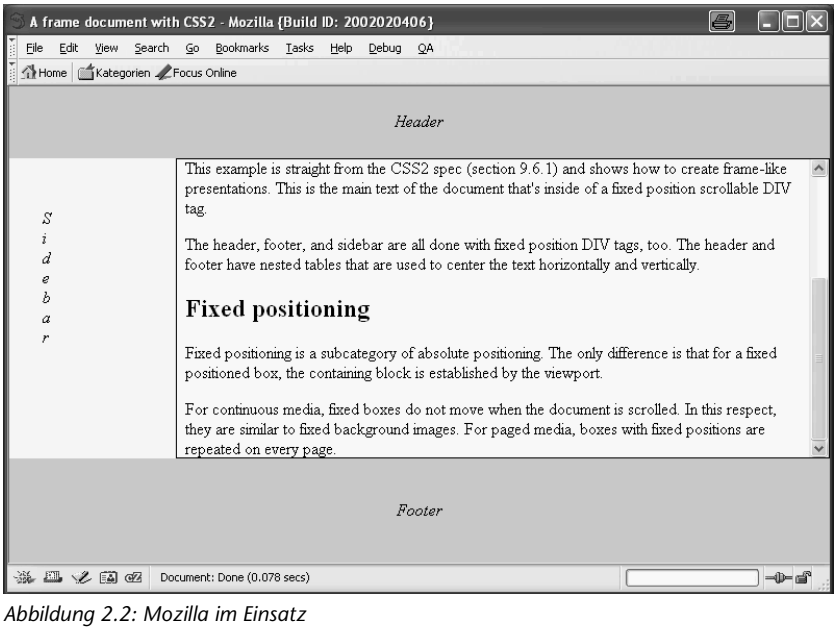


Abbildung 2.2: Mozilla im Einsatz

Die Abbildung 2.2 zeigt den Browser Mozilla. Hierbei handelt es sich um ein Open- Source-Produkt, an dem zahlreiche Entwickler weltweit beteiligt sind. Dieser Browser orientiert sich stark an bestehenden Standards und bietet somit eine gute Möglichkeit, JavaScript-Programme zu testen. Mozilla kann unter *http://www.mozilla.org* heruntergeladen werden.

2.1.3 Script-Debugger

automatisierte Fehlersuche

Sie werden während der Entwicklung von JavaScript-Programmen häufig Fehlermeldungen angezeigt bekommen. An dieser Stelle sollen Software-Produkte vorgestellt werden, die dabei, helfen JavaScript-Fehler zu beheben. JavaScript-Fehler erkennen Sie zumeist erst dann, wenn die Seite im WWW-Browser aufgerufen wird. Wie Script-Fehler vom Browser angezeigt werden, hängt von dem verwendeten Produkt und den vorgenommenen Einstellungen ab. So lassen sich im Netscape Navigator und im Mozilla-Browser Fehlermeldungen in der JavaScript-Console anzeigen. Um diese aufzurufen, müssen Sie innerhalb der Adresszeile die Anweisung *javascript:* eingeben. Daraufhin wird die JavaScript-Console gestartet.

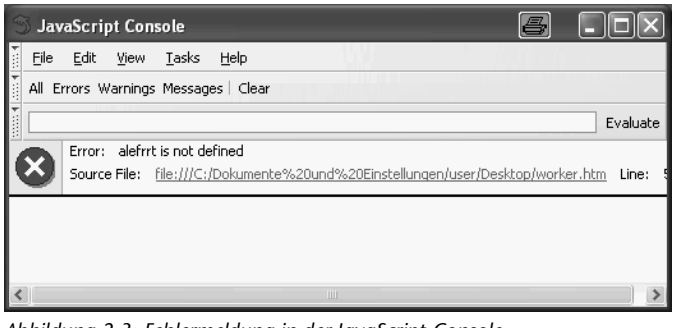


Abbildung 2.3: Fehlermeldung in der JavaScript-Console

Wie Abbildung 2.3 zeigt, werden die eventuell vorhandenen Fehler angezeigt. So kann der Browser in dem gezeigten Beispiel die Methode `alefrrt` nicht zuordnen und somit das JavaScript-Programm nicht ausführen. Der Grund für die Fehlermeldung ist hier, dass die Methode `alefrrt` nicht definiert ist. An deren Stelle sollte eigentlich die **`alert()`**-Methode notiert werden. Ein Schreibfehler führte hier also zu einer Fehlermeldung. Nun ist es freilich nicht so, dass der Grund für die Fehlermeldung immer so offensichtlich ist. Besonders bei komplexen Programmen kommt es beispielsweise häufig vor, dass Variablen innerhalb des Programmflusses ungültige Werte annehmen. In einem solchen Fall wäre die Fehlersuche weitaus schwieriger als bei einem bloßen Schreibfehler. Um komplexeren Problemen auf die Spur zu kommen, empfiehlt sich der Einsatz von Script-Debuggern.

Software zur Fehleranalyse

Script-Debugger sind spezielle Programme, die der Fehleranalyse von JavaScript-Programmen dienen. Im Laufe dieses Abschnitts finden Sie eine Übersicht der drei am häufigsten eingesetzten Script-Debugger sowie die hierzu relevanten Downloadmöglichkeiten. Exemplarisch und anhand der Abbildung 2.4 soll der Microsoft Script Debugger vorgestellt werden. Dieser integriert sich nach dem Download und der Installation direkt in den Internet Explorer. Treten innerhalb eines Dokuments Fehler auf, wird dieser automatisch gestartet. Innerhalb des Debuggers können Sie nun den Programmfluss des fehlerhaften JavaScripts verfolgen.

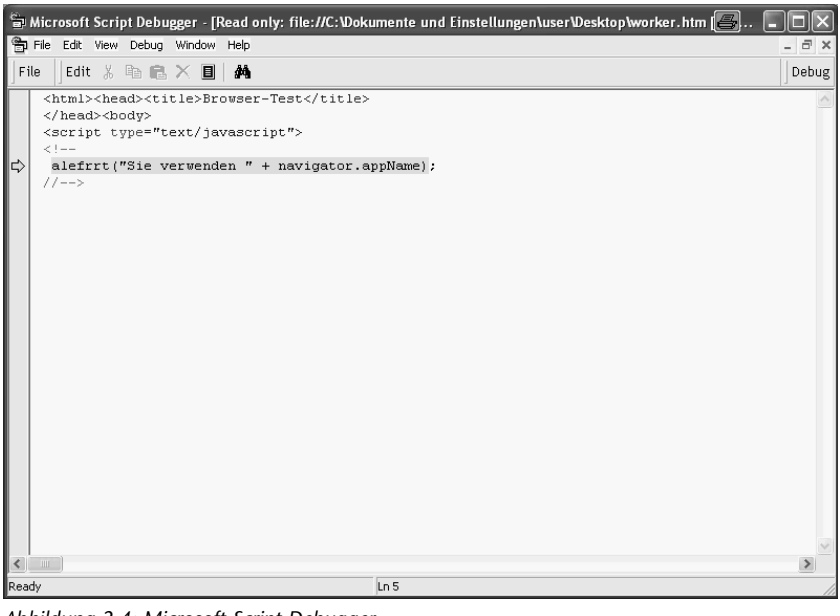


Abbildung 2.4: Microsoft Script Debugger

Alle Script-Debugger funktionieren auf ähnliche Weise. Die Unterschiede sind zumeist nur marginal. So ist der Script-Debugger von Mozilla beispielsweise eine Java-Anwendung, die über den Browser gestartet werden muss. Aufgerufen wird dieser über TASKS TOOLS JAVASCRIPT DEBUGGER. Die folgende Tabelle soll Ihnen bei der Suche nach einem geeigneten Script-Debugger behilflich sein. Selbstverständlich existieren weitere Debugger. Diese sind jedoch zumeist nicht kostenlos erhältlich und eignen sich somit vornehmlich für professionelle Programmierer.

Name	Download-Adresse
Microsoft Script Debugger	http://msdn.microsoft.com/downloads/default.asp?URL=/downloads/sample.asp?url=/msdn-files/027/001/731/msdncompositedoc.xml
Netscape JavaScript Debugger	http://developer.netscape.com/software/jsdebug.html
Mozilla Debugger	http://www.mozilla.org/projects/venkman/

Tabelle 2.1: Script-Debugger und Downloadmöglichkeiten

Sie sollten auf Ihrem System in jedem Fall einen Script-Debugger installieren. Zu groß sind die hieraus resultierenden Vorteile. Besonders Neueinsteigern sei die Verwendung eines Script-Debuggers bereits während der Programmierung der ersten JavaScript-Programme ans Herz gelegt.

2.1.4 Problemanalyse

Neben der Auswahl der Werkzeuge, wie Browser und Texteditor, sollte vor dem Einsatz von JavaScript überprüft werden, ob diese Sprache für die aktuellen Anforderungen geeignet ist. Zwar lassen sich mit JavaScript viele Probleme lösen, aber nicht für alle ist es die geeignetste Sprache. So lassen sich zwar durchaus Suchmaschinen auf JavaScript-Basis erstellen, für diesen Zweck eignen sich jedoch andere Sprachen besser. Als Faustregel wollen wir Folgendes festhalten: JavaScript sollte nur dann Verwendung finden, wenn das bestehende Problem am effizientesten mit dieser Sprache gelöst werden kann. Einschränkungen hierfür gelten freilich immer dann, wenn die Kenntnisse in anderen Sprachen nicht vorhanden sind. In einem solchen Fall wäre es dann sicherlich sinnvoller, das Problem mit einem eleganten JavaScript anstatt mit einem fehlerhaften Perl-Script zu lösen.

2.1.5 HTML

Grundkenntnisse vorausgesetzt

Ohne HTML-Kenntnisse lassen sich die Beispiele in diesem Buch nicht oder nur schwer nachvollziehen. Sie müssen nun sicherlich keine Koryphäe in HTML sein, dennoch ist es von elementarer Bedeutung, dass Ihre Kenntnisse so weit reichen, dass die grundlegende Syntax bekannt ist. So sollten Sie die Begriffe Tag und Attribut in jedem Fall kennen und verstehen. Zudem wäre es von Vorteil, wenn die gängigsten HTML-Elemente bekannt sind. All die hier genannten Einschränkungen gelten allerdings nur für den Einsatz von JavaScript. Das Erlernen der Sprache und ihrer Syntax lässt sich jedoch auch ohne HTML-Kenntnisse realisieren.

2.1.6 Programmierverständnis

Einstieg in andere Programmiersprachen

Um JavaScript zu erlernen, sind Kenntnisse in anderen Programmiersprachen nicht zwingend erforderlich. Vielmehr eignet sich JavaScript vorzüglich dazu, einen Einstieg in die Welt der Programmierung zu erlangen. Denn ebenso wie in anderen Programmiersprachen existieren auch in JavaScript solche Sprachelemente wie Schleifen und bedingte Anweisungen. Problematisch an JavaScript ist indes, dass wichtige Elemente von Programmiersprachen bislang fehlen. So ist es beispielsweise nicht möglich, eine implizite Typkonvertierung vorzunehmen. Programmierer, die andere Sprachen beherrschen, werden einen schnellen Zugang zu JavaScript finden. So wird ein Java-Programmierer viele Syntaxelemente wiederfinden, die ihm von seiner „Haussprache" bekannt sind.

Grundlegend sei an dieser Stelle festgehalten, dass für die Verwendung von JavaScript Programmierkenntnisse nicht zwingend vorhanden sein müssen. Zwar würde dies den Einstieg in diese Sprache erleichtern, ein ähnlicher Effekt lässt sich allerdings auch bei Nicht-Programmieren erzielen. Wichtig ist lediglich, dass ein Interesse an dieser Sprache besteht und mit der Zeit die Prinzipien von JavaScript verstanden werden. Ist diese Hürde genommen, lassen sich JavaScript-Konzepte nicht nur für den Einsatz auf Bilderwechsel im WWW, sondern auch als Einstieg in andere Programmiersprachen verwenden.

2.2 JavaScript integrieren

*type **statt** language*

Ein clientseitiger Einsatz von JavaScript-Programmen erfordert die Einbettung des Quellcodes in eine HTML-Datei. Wird die aktuelle Seite vom Browser aufgerufen, lädt dieser das JavaScript und führt es anschließend aus. Um dieses Vorgehen tatsächlich anwenden zu können, muss der Browser zwischen JavaScript-Code und anderen Elementen wie beispielsweise simplen HTML-Textelementen unterscheiden. Aus diesen Gründen muss Programmcode stets gesondert gekennzeichnet werden. Hierfür wird das `<SCRIPT>`-Tag verwendet. Die hiermit deklarierten Zeilen werden nicht am Bildschirm ausgegeben, sondern als Programmcode interpretiert. Dem WWW-Browser sollte explizit mitgeteilt werden, welche Script-Sprache für die Programmierung der entsprechenden Seite verwendet wird. Für diesen Zweck steht das `type`-Attribut, welches dem `<SCRIPT>`-Tag zugewiesen wird, zur Verfügung. Als Wert wird hierbei, handelt es sich tatsächlich um die in diesem Buch vorherrschende Sprache, JavaScript zugewiesen. Es existieren keinerlei feste Regeln, an welcher Stelle einer HTML-Datei ein JavaScript-Bereich definiert werden muss. In der Praxis hat sich mittlerweile jedoch die JavaScript-Notation innerhalb des `<HEAD>`-Bereichs als effizienter und übersichtlicher Programmierstil herauskristallisiert.

Durch diese Code-Behandlung kann am ehesten, und dies gilt insbesondere für ladeintensive Seiten, gewährleistet werden, dass der Browser den JavaScript-Code bereits dann eingelesen hat, wenn dieser ausgeführt werden soll. Wie es nun mittlerweile guter Brauch ist, soll die Integration eines JavaScripts anhand des allseits beliebten „Hallo-Welt"-Programms demonstriert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
alert("Hallo Welt")
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 2.1: Hallo Welt durch JavaScript

Innerhalb des einleitenden **<SCRIPT>**-Tags ist es seit HTML 4 vorgeschrieben, den MIME-Typ der verwendeten Script-Sprache zu benennen. Hierzu werden das Attribut **type** sowie für JavaScript der MIME-Typ **text/javascript** verwendet. Häufig wird im einleitenden **<SCRIPT>**-Tag zusätzlich das **language**-Attribut eingesetzt.

Hierauf kann und sollte verzichtet werden. Zum einen steht dieses Attribut auf der Streichliste des W3C und zum anderen kann dieses nicht von allen Browsern korrekt interpretiert werden. Im nächsten Schritt sollte gewährleistet werden, dass Browser, die nicht JavaScript-fähig sind, den integrierten Script-Bereich übergehen. Zu diesem Zweck wird der JavaScript-Quellcode in einen Kommentar eingeschlossen.

Der WWW-Browser führt die Zeilen, die sich innerhalb des **<SCRIPT>**-Tags befinden, aus. Hiervon ausgenommen müssen Funktionsdefinitionen betrachtet werden, die erst nach dem Aufruf einer Funktion interpretiert werden. Und eben aus diesem Grund empfiehlt sich die Definition aller JavaScript-Elemente innerhalb des Dateikopfes. Neben der gezeigten Notationsvariante lassen sich innerhalb einer HTML-Datei mehrere Script-Bereiche integrieren. Für diesen Zweck muss lediglich eine Abfolge mehrerer **<SCRIPT>**-Tags notiert werden. Hierbei ist auf eine hierarchische Tag-Strukturierung, wie diese auch in HTML üblich ist, zu achten. Selbstverständlich existieren Ausnahmen bezüglich der zuvor aufgeführten Aussagen. So können **<SCRIPT>**-Bereiche, besonders wenn diese der dynamischen Erzeugung von bestimmten Angaben dienen

sehr wohl konsequenterweise innerhalb des <BODY>-Tags notiert werden. Dies gilt vor allem dann, wenn bestimmte Angaben dynamisch erzeugt werden sollen. Als ein hierfür vakantes Beispiel soll die Anzeige der aktuellen Uhrzeit am Ende der Datei dienen. In diesem Fall bietet sich als Container für das JavaScript der <BODY>-Bereich nahezu an. Hierzu aber an geeigneterer Stelle mehr.

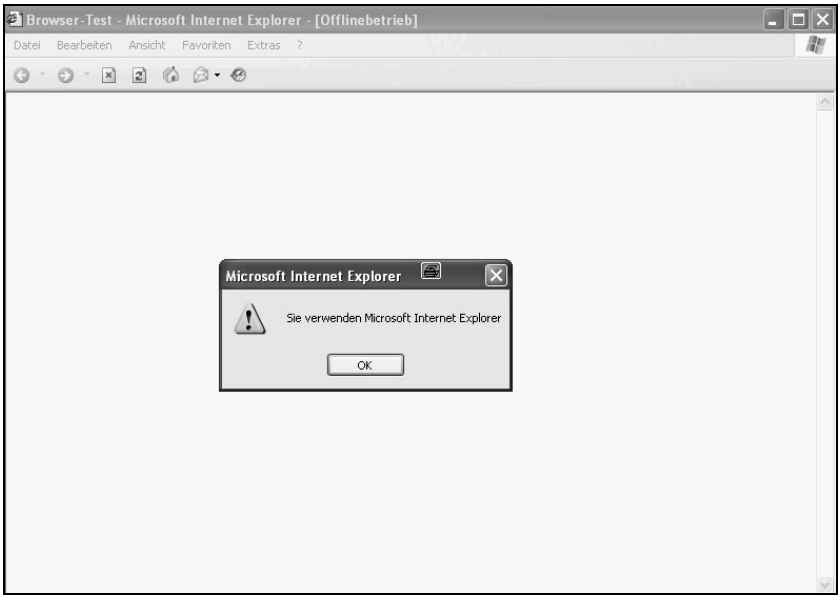


Abbildung 2.5: Informationen in einem Meldungsfenster

Nicht alle Browser beherrschen JavaScript. Für Programmierer kommt erschwerend hinzu, dass einige Anwender die JavaScript-Funktionalität ihres Browser deaktivieren. Mehr zu diesem Thema finden Sie in Abschnitt 2.8.1. Hier gibt es auch eine Erörterung über das Notieren des JavaScript-Blocks innerhalb einer auf den ersten Blick als SGML-Kommentar erscheinenden Schreibweise.

2.2.1 Weiterführendes zu <script>

vergessene Attribute

Das <SCRIPT>-Tag bietet mehr Variationsmöglichkeiten, als dies innerhalb der bisherigen Abhandlung suggeriert wurde. Einige Attribute mögen dem Leser bereits bekannt sein, andere aber kommen in der Praxis seltener zur Anwendung und sind somit fast in Vergessenheit geraten. Um diese wieder in das Bewusstsein von JavaScript-Programmierern zurückzuholen, werden in diesem Abschnitt alle Attribute, die sich auf das <SCRIPT>-Tag anwenden lassen, vorgestellt.

charset

Zeichensatz

Das **charset**-Attribut spezifiziert den innerhalb einer externen Datenquelle verwendeten Zeichensatz. Sinnvoll ist der Einsatz dieses Attributs vor allem im Zusammenhang mit der Erstellung von multinationalen Seiten, bei denen auf verschiedene sprachraumspezifische Sonderzeichen zurückgegriffen werden muss. Die folgende Tabelle beschreibt die wichtigsten Sprachstandards mit den Zeichensätzen, für welche diese vorgesehen sind.

Standard	Zeichensatz
ISO-8859-1	Westeuropa, Lateinamerika
ISO-8859-2	Osteuropa
ISO-8859-3	Südeuropa
ISO-8859-4	Skandinavien, Baltikum
ISO-8859-5	Kyrillisch
ISO-8859-6	Arabisch
ISO-8859-7	Griechisch
ISO-8859-8	Hebräisch
ISO-8859-9	Türkisch
ISO-8859-10	Lappland
EUC_JP	Japanisch 1
Shift_JIS	Japanisch 2
ISO-2022-JP	Japanisch 3

Tabelle 2.2: Zeichensätze und Sprachstandards

Für den westeuropäischen Sprachraum, also auch für Deutsch, wird der „ISO-8859-1"-Standard eingesetzt. Da die Mehrzahl der Leser dieses Buches aus diesem Raum stammen dürfte, zielt das folgende Beispiel auf diese Bedürfnisse ab. Dieser Quellcode-Auszug beschreibt die Integration einer externen Datenquelle mit gleichzeitiger Bekanntgabe des dort verwendeten Sprachstandards.

```
<SCRIPT type="text/JavaScript" charset="ISO-8859-1"
src="extern.js">
</SCRIPT>
```

Vor dem Einsatz des **charset**-Attributs innerhalb des **<SCRIPT>**-Tags ist zu bedenken, dass eine solche Auszeichnung tatsächlich nur dann Sinn macht, wenn auf eine externe Datenquelle zugegriffen wird. Anderenfalls, also wenn sich der JavaScript-Quellcode direkt innerhalb der aktuellen Datei befindet, wird die **charset**-Anweisung als Meta-Angabe notiert. Dieses Vorgehen ist so auch in HTML üblich.

defer

Automatismen unterbinden

Seit HTML 4 steht das **defer**-Attribut innerhalb des **<SCRIPT>**-Tags zur Verfügung. Durch dessen Einsatz ist es zumindest theoretisch möglich, das automatische Generieren von Inhalten durch JavaScript zu unterbinden. Der Internet Explorer beherrscht diese Syntax seit der Produktversion 4. Netscape ging indes bislang noch nicht auf diese Erweiterung ein und führt Befehle wie beispielsweise **documnt.write()** trotz des **defer**-Attributs augenblicklich aus.

```
<SCRIPT type="text/JavaScript" defer>
< !--
document.write("Hallo Welt");
//-->
</SCRIPT>
```

In diesem Beispiel soll der Text `Hallo Welt` automatisch generiert werden. Für den Fall, dass der Internet Explorer auf dieses Script stößt, geschieht aber zunächst nichts dergleichen. Ausgelöst werden kann die gewünschte Aktion jedoch anschließend durch weiterführende Script-Anweisungen. Die Vorteile des **defer**-Attributs mögen sich zugegebenermaßen nicht auf den ersten Blick erschließen und dennoch existieren sie. So können beispielsweise Dokumente durch das nicht sofortige Ausführen von **document.write()**-Anweisungen vom Browser schneller dargestellt und dem Nutzer somit binnen kürzester Zeit die entsprechenden Inhalte präsentiert werden.

event und for

Ereignisse bekannt geben

Bislang werden die beiden Attribute **event** und **for** lediglich vom Internet Explorer ab Produktversion 3.x interpretiert. Ausdrücklich spezifiziert wurden beide Attribute bislang nicht innerhalb des HTML-, dafür jedoch innerhalb des DOM-Standards nach Level 2. Beide Attribute sind stets in einem Kontext zu betrachten, da nur das Notieren beider Angaben zu dem gewünschten Ergebnis führt. Da auf die praktische Anwendung dieser Syntaxform innerhalb des Abschnitts Fehler! Verweisquelle konnte nicht gefunden werden. noch eingegangen wird, soll hier lediglich ein Kurzabriss mit den notwendigsten Informationen erfolgen. Das folgende Beispiel beschreibt eine typische Anwendung beider Attribute und zeigt, wie sich ein JavaScript direkt auf ein spezielles Ereignis, in diesem Fall ist dies **onclick**, anwenden lässt.

```
<HTML>
<HTML>
<SCRIPT type="text/JavaScript" event="onclick" for="public">
<!--
```

```
alert ("Spezielles im Internet Explorer");
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT type="button" value="anklicken" name="public">
</FORM>
</BODY>
</HTML>
```

Listing 2.2: Eine `alert()`-Meldung im Internet Explorer

In diesem Beispiel wird durch das Anklicken des Buttons, dem der interne Bezeichner `public` zugewiesen wurde, eine **`alert()`**-Box ausgegeben. Erreicht wird dies auf Grund der Symbiose der beiden Attribute **`event`** (dem hier der Event-Handler **`onclick`** zugewiesen wurde) und **`for`** (das den Namen des Buttons zugewiesen bekommt). Vor dem Einsatz der gezeigten Syntax ist in jedem Fall die fehlende Interpretation anderer Browser als dem Microsoft Internet Explorer zu berücksichtigen.

id

Objekte eindeutig kennzeichnen

Das Attribut **`id`** ist in vielerlei Hinsicht sinnvoll. So kann durch dieses Attribut jedes Objekt eindeutig identifiziert und somit im Nachhinein auf dieses zugegriffen werden. An welcher Stelle das **`id`**-Attribut eingesetzt wird, ist indes nicht genau spezifiziert. So können Programmierer einfache HTML-Elemente als dateiweit eindeutig auszeichnen. Ebenso ist es aber auch möglich das **`id`**-Attribut innerhalb des **`<SCRIPT>`**-Tags zu notieren, um darauf zugreifen zu können.

```
<HTML>
<HTML>
<SCRIPT type="text/JavaScript" event="onclick" for="public">
<!--
alert ("Spezielles im Internet Explorer");
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT type="button" value="anklicken" id="public">
</FORM>
</BODY>
</HTML>
```

Listing 2.3: Die Verwendung des `id`-Attributs

Die aufgeführte Syntax beschreibt eine Kombination aus den verschiedensten Attributen. Auf Grund der speziellen Anforderungen des Beispiels wird es notwendig, das HTML-Element `<INPUT>` durch das Zuweisen des `id`-Attributs innerhalb der Datei als eindeutig zu referenzieren.

language

verwendete Script-Sprache angeben

Es existieren verschiedene Versionen von JavaScript und, was erschwerend hinzukommt: Internetseiten können eben nicht nur durch die zahlreichen JavaScript-Varianten sondern auch unter Zuhilfenahme anderer Sprachen wie beispielsweise VB-Script programmiert werden. Bei diesen beiden Problemstellungen sollte stets das `language`-Attribut des `<SCRIPT>`-Tags zu deren Lösung herangezogen werden. So kann durch dessen Einsatz neben der innerhalb der aktuellen Seite verwendeten Script-Sprache auch die entsprechende Sprachversion explizit ausgezeichnet werden. Erst hierdurch wird es möglich, dass der Browser bei einer Nichtimplementierung einer bestimmten Script-Sprache oder von Sprachstandards den Inhalt des `<SCRIPT>`-Tags nicht ausliest und somit Fehlermeldungen vermieden werden können. Da sich dieses Buch fast ausschließlich mit JavaScript beschäftigt, sollen der Einsatz und die Möglichkeiten des `language`-Attributs auch anhand dieser Sprache verdeutlicht werden. Die folgende Tabelle zeigt alle bis zum jetzigen Zeitpunkt entwickelten JavaScript-Sprachstandards mit dem entsprechenden Wert des `language`-Attributs.

Wert	Bedeutung
JavaScript	Alle Browser, die JavaScript implementieren
JavaScript1.1	Alle Browser, die mindestens JavaScript 1.1 implementieren
JavaScript1.2	Alle Browser, die mindestens JavaScript 1.2 implementieren
JavaScript1.3	Alle Browser, die mindestens JavaScript 1.3 implementieren
JavaScript1.4	Alle Browser, die mindestens JavaScript 1.4 implementieren
JavaScript1.5	Alle Browser, die mindestens JavaScript 1.5 implementieren

Tabelle 2.3: Sprachstandards und Browser-Unterstützung

Die beiden „großen“ Browser, also sowohl der Internet Explorer wie auch der Netscape Navigator, haben standardmäßig als Script-Sprache JavaScript eingestellt. Diese Aussage ließe nun darauf schließen, dass, handelt es sich bei der eingesetzten Sprache tatsächlich um JavaScript, keine explizite Auszeichnung über die Angabe `language="JavaScript"` von Nöten ist. Mag diese Browserbehandlung auch teilweise korrekt sein, können Fehlinterpretationen dennoch nicht gänzlich ausgeschlossen werden.

Seit HTML 4 sollte für die Kennzeichnung der verwendeten Sprache allerdings nicht mehr das **language**-, sondern das **type**-Attribut eingesetzt werden. Der Vollständigkeit halber folgt ein Beispiel, welches die Anwendung des **language**-Attributs zeigt:

```
<SCRIPT language="JavaScript1.3">
document.write("Implemntierung von JavaScript1.3")
</SCRIPT>
```

Dieses kleine Script beschreibt den Einsatz des **language**-Attributs. Ein hierdurch ausgezeichnetes JavaScript sollte sich auf Grund des eingesetzten Wertes auch tatsächlich am JavaScript-Sprachstandard-1.3 orientieren und dessen Regeln korrekt einhalten. Trifft ein WWW-Browser auf dieses Script und versteht dieser die Version 1.3 von JavaScript nicht, übergeht er diese Datei-Passage. Diese Sprachversion implementierende Browser können indes auf den Inhalt des Scripts zurückgreifen und dessen Inhalt ausführen.

Zwar sind die Aussagen in diesem Abschnitt korrekt, die Browserinterpretation ist in einigen Fällen jedoch fehlerhaft. So geben zwar einige WWW-Browser vor, bestimmte Sprachstandards zu verstehen, in der Praxis erweisen sich diese Aussagen jedoch häufig als falsch. Demzufolge sollte die Kontrolle der tatsächlichen Funktionalität des Scripts auch nach wie vor auf der Seite des Programmierers bleiben.

src

externe JavaScript-Programme

Auf das **src**-Attribut wird zwar noch genauer eingegangen, dennoch soll dessen Bedeutung hier der Vollständigkeit halber bereits kurz vorgestellt werden. Durch das **src**-Attribut wird es möglichn JavaScript-Quellcode aus der eigentlichen Datei auszulagern. Die Gründe für eine solche Auslagerung können vielfältiger Natur sein. So macht dies beispielsweise immer dann Sinn, wenn der Umfang des Scripts sehr groß ist und somit die Lesbarkeit und die Übersichtlichkeit der eigentlichen Container-Datei nicht mehr gewahrt werden könnten. Aber auch im Hinblick auf eine Minimierung der Ladezeiten kann das **src**-Attribut eine entscheidende Rolle spielen. Für weiterführende Informationen lesen Sie bitte im Abschnitt 2.7 nach.

```
<SCRIPT type="text/javascript" src="../menu.js">
</SCRIPT>
```

Für die Spezifizierung einer externen JavaScript-Datei müssen deren Name sowie gegebenenfalls deren Dateipfad angegeben werden. Bei der Auslagerung von JavaScript-Quellcode gelten die gleichen Regeln, die auch im Hinblick auf das Einbinden von Grafiken in HTML ihre Gültigkeit besitzen.

type

MIME-Typ bekannt geben

Das **type**-Attribut dient der Bekanntgabe des verwendeten MIME-Typs. Vergleichbar ist die **type**- mit der **language**-Anweisung. Es ist lediglich darauf zu achten, dass, anders als bei **language,** lediglich der MIME-Typ, jedoch nicht die verwendete Sprachversion angegeben wird. Die folgende Tabelle zeigt häufig verwendete Typ-Bezeichnungen innerhalb des **<SCRIPT>**-Tags.

Typ	Bedeutung
text/javascript	JavaScript
text/jscript	Jscript
text/php	PHP
text/vbscript	Microsoft Visual Basic Script

Tabelle 2.4: MIME-Typen

Ein kurzer Hinweis zur Browserinterpretation: Das **type**-Attribut gehört seit der HTML-Version 4 zum offiziellen Sprachstandard. Der Internet Explorer kann das **type**-Attribut seit der Produktversion 4.x interpretieren. Anders verhält sich dies, so zumindest die Verlautbarung seitens Netscapes, im Zusammenhang mit dem Navigator. Dieser sollte das **type**-Attribut bis einschließlich Netscape 6 ignorieren. Dass dieses Verhalten sich in der Praxis wiederlegen lässt, ist jedoch mittlerweile ein offenes Geheimnis. Denn bei sich wiedersprechenden Angaben zu **language** und **type** greift der Netscape Navigator dennoch auf die aus seiner Sicht korrekte Anweisung zurück. Dies gilt auch, wenn dem **language**-Attribut ein fehlerhafter Wert zugewiesen, dafür aber der MIME-Typ über die Anweisung **type** korrekt spezifiziert wurde.

```
<SCRIPT type="text/JavaScript">
<!--
Script-Inhalt
//-->
</SCRIPT>
```

Die Verwendung des **type**-Attributs erfordert keine besonderen Programmierfähigkeiten. Das **type**-Attribut wird genauso wie in HTML eingesetzt. Bei der Wertzuweisung spielt es keine Rolle, ob hier Groß- oder Kleinschreibung bevorzugt verwendet wird. Aus Gründen einer einheitlichen Darstellung des Quellcodes sollte aber dennoch eine Schreibweise konsequent verwendet werden.

Meta-Tags als Alternative

Eine weitere Möglichkeit zur Bestimmung der verwendeten Scriptsprache steht durch den Einsatz von Meta-Angaben zur Verfügung. Eingesetzt werden sollte eine diesbezügliche Anweisung immer dann, wenn innerhalb einer Datei mehrere Scripts angewandt und diese nicht in jedem Fall als einzelne Script-Bausteine ausgezeichnet werden sollen. Für JavaScript stellt sich die Notation einer solchen Meta-Anweisung folgendermaßen dar:

```
<HEAD>
<META HTTP-EQUIV="CONTENT-SCRIPT-TYPE" CONTENT="text/javascript">
</HEAD>
```

Dem Attribut **content** wird der bevorzugte MIME-Typ der verwendeten Script-Sprache zugewiesen. Für JavaScript ist dies **text/javascript**. Wird eine andere Script-Sprache als vorherrschend deklariert, muss der Wert des **content**-Attributs entsprechend angepasst werden.

2.2.2 Externe JavaScript-Programme

ein Script für mehrere Dateien

Die direkte Integration von JavaScript-Quellcode in eine HTML-Datei kann, so korrekt diese Programmbehandlung auch sein mag, in einigen Fällen die Anforderungen eines Internetauftritts nur ungenügend erfüllen. Selbstverständlich ist das Notieren eines JavaScript-Programms innerhalb der HTML-Datei völlig korrekt. Für einige Anwendungen erscheint das Auslagern des Programmcodes jedoch das probatere Mittel zu sein. So führt diese Script-Behandlung beispielsweise zu einer erhöhten Übersichtlichkeit des Quellcodes. Aber auch andere Aspekte sprechen für eine Auslagerung von JavaScript in eine separate Datei. So kommt es während der Umsetzung eines WWW-Projekts häufig zur mehrmaligen Verwendung eines Scripts auf mehreren Seiten. Bedient sich der Programmierer nicht der Möglichkeit der Script-Auslagerung, sondern integriert er das JavaScript direkt in die HTML-Seite, muss er dies für jede HTML-Datei tun. Durch die Script-Auslagerung wird es indes möglich, lediglich eine JavaScript-Datei zu erstellen und auf diese durch eine jeweilige Referenzierung aus allen HTML-Dateien zuzugreifen. Wie sich die Auslagerung eines JavaScripts darstellt, soll anhand des bereits zuvor aufgeführten „Hallo Welt“-Programms veranschaulicht werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript" src="script.js">
<!--
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 2.4: Ein Verweis auf ein externes JavaScript-Programm

Es ist auf den ersten Blick zu erkennen, dass die gezeigte HTML-Datei zwar ein **<SCRIPT>**-Tag enthält, die Suche nach entsprechendem Programmcode hier jedoch nicht von Erfolg gekrönt sein wird. Es befindet sich, neben herkömmlichem HTML-Code, lediglich eine Referenz auf eine externe Datei darin. Um eine solche Referenzierung umsetzen zu können, muss innerhalb des **<SCRIPT>**-Tags zunächst bekannt gegeben werden, welcher Sprache sich die externe Datei bedient. Zu diesem Zweck wird dem Attribut, so es sich auch tatsächlich um ein JavaScript handelt, das Attribut **type** und als dessen Wert **text/JavaScript** hinzugefügt. Das **src**-Attribut spezifiziert die Quelle des einzubindenden JavaScripts. Bei dieser Datei, auf welche diese Angabe abzielt, handelt es sich um eine reine ASCII-Datei, in der nichts weiter als JavaScript-Quellcode enthalten sein sollte. Als deren Suffix wird *.js empfohlen. Für das Einbinden des JavaScript, via **src** gelten im Übrigen die gleichen Regeln wie für das Einbinden von Grafiken in HTML und die hierbei relevanten Bedingungen für Pfadangaben etc.

Um auf das „Hallo Welt"-Beispiel zurückzukommen: Der Inhalt der nun ausgelagerten Datei `script.js` stellt sich folgendermaßen dar:

```
alert("Hallo Welt")
```

Es ist zu erkennen, dass in dieser Datei nichts als reiner JavaScript-Quellcode notiert ist. Sollten hinsichtlich der Ausführung des Scripts wegen der Auslagerung Probleme auftauchen, liegt dies, wenn denn das Script korrekt ist, zumeist an der Konfiguration des WWW-Servers. Kann das Script also aus Gründen des externen Zugriffs nicht ausgeführt werden, sollte das Vorhandensein des **text/javascript**-MIME-Typs in der Konfiguration des WWW-Servers überprüft und gegebenenfalls hinzugefügt werden.

Externe Dateien und Server-Statistiken

Ausgelagerte JavaScript-Programme bieten sicherlich zahlreiche Vorteile. So wird der Quellcode der HTML-Datei nicht mit Script-Bereichen unübersichtlich gemacht und die reine HTML-Datei lässt sich effizienter bearbeiten. Externe JavaScript-Dateien bieten jedoch auch Nachteile. Am gravierendsten wirken sich diese auf Server-Statistiken aus. Zum Verständnis: Jede externe Datei, die von einer HTML-Seite aufgerufen wird, wird auch tatsächlich vom Server jedes Mal geladen. Dies bedeutet, dass sich hieraus ein verfälschtes Bild der aufgerufenen Dateien ergibt. So können Sie davon ausgehen, dass eine externe JavaScript-Datei, die von mehreren HTML-Dateien genutzt wird, in der Server-Statistik einen der vorderen Plätze einnimmt. Dass sich dann hieraus keine Rückschlüsse auf das tatsächliche Nutzerverhalten des Internetprojekts ziehen lassen, versteht sich von selbst. Erschwerend kommt hinzu, dass ja eben nicht nur JavaScript-, sondern auch Style-Sheet-Dateien ausgelagert werden können. Die Statistik wird demnach mehr und mehr ungenau und kann nur noch bedingt zur Analyse der Seite verwendet werden. Sie sollten die hier aufgeführten Punkte in jedem Fall vor der Planung einer Internetseite berücksichtigen und die Vor- und Nachteile externer JavaScript-Dateien gegeneinander abwägen.

2.2.3 JavaScript in HTML-Tags

Event-Handler

JavaScript-Bereiche umfassen weit mehr als die alleinige Definition innerhalb des Dateikörpers. So besteht eine häufig eingesetzte Variante beispielsweise darin, dass ein herkömmliches HTML-Tag für das Auslösen eines JavaScripts genutzt wird. Speziellere Anwendungsformen werden im Laufe dieses Buches noch zur Genüge vorgestellt. Diese Einführung an früherer Stelle dient vielmehr der Abrundung und Vervollständigung aller geläufigen JavaScript-Aufrufe. Um JavaScript-Funktionen, Methoden oder Objekte aus einem HTML-Tag heraus aufrufen zu können, wird sich der so genannten Event-Handler bedient. Speziellere Informationen zu deren Aufruf und Einsatzmöglichkeiten finden Sie im Abschnitt 3.9. Hier soll ein kleines Beispiel zumindest einen Einstieg in diese Materie ermöglichen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function drucken()
{
if (window.print)
      window.print()
```

```
else
 window.alert("Das geht leider nicht!")
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:drucken()">Ausdrucken der Seite</A>
</BODY>
</HTML>
```

Listing 2.5: Aufruf der `druck()`-Funktion

Das Beispiel zeigt, dass sich die grundlegende Syntax der HTML-Datei mit integriertem JavaScript-Bereich unverändert präsentiert. So wird auch hier in dem **<HEAD>**-Tag eine JavaScript-Funktion integriert. Deren Name lautet `drucken( )` und wird mittels eines Hyperlinks aufgerufen. Hierzu wird dem **href**-Attribut das Schlüsselwort **javascript** zugewiesen. Nach dem sich hieran anschließenden Doppelpunkt folgt der Funktionsname `drucken()`.

2.2.4 No-Script-Bereiche

Nicht-JavaScript-fähige Browser

Gängige WWW-Browser beherrschen in der Regel JavaScript oder andere Script-Sprachen. Gleichwohl sind aber auch Produkte auf dem Markt, welche diese Eigenschaft nicht besitzen. Aber nicht nur für die Anwender solcher Browser ist die in diesem Abschnitt vorgestellte Syntax von Bedeutung. So kann beispielsweise jeder Anwender den von ihm verwendeten Browser so konfigurieren, dass dieser keinen JavaScript-Code mehr ausführt bzw. erkennt. Trifft ein mit einem solchem Browser ausgestatteter Anwender auf einen mit JavaScript gestalteten Internetauftritt, stehen sowohl der Anwender als auch der Entwickler vor einem Dilemma. Dieses kann sich sogar noch steigern, wenn die gesamte Navigation des WWW-Projekts über JavaScript gesteuert wird. Um all diesen Widrigkeiten entgegenzuwirken und geeignete Gegenmaßnamen zu ergreifen, steht das **<NOSCRIPT>**-Tag bereit. Vergleichbar sind dessen Einsatz und Wirkungsweise mit denen des **<NOFRAMES>**-Tags, welches ja bekanntlich für WWW-Browser konzipiert wurde, die keine Frames darstellen können. Trifft ein Browser auf eine framegesteuerte Seite, liest er den Inhalt aus, welcher innerhalb des **<NOSCRIPT>**-Tags notiert wurde und ignoriert die framespezifischen Anweisungen. Im Zusammenhang mit dem **<NOSCRIPT>**-Tag verhält es sich ähnlich. Trifft ein Anwender auf eine WWW-Seite, die JavaScript enthält, und kann dessen Browser diesen Programmcode nicht interpretieren, werden lediglich die innerhalb des **<NOSCRIPT>**-Tags bereitgestellten Informationen

angezeigt. So kann in diesem Bereich beispielsweise eine alternative Version des Internetauftritts oder, wenn dies ein nicht mehr zumutbares Mehr an Aufwand bedeutet, so doch zumindest ein hinweisender Text auf die von Ihnen verwendete Technologie JavaScript notiert werden. Die Verwendung des **<NOSCRIPT>**-Tags stellt sich folgendermaßen dar:

```
<SCRIPT type="JavaScript">
Inhalt des JavaScript-Programms
</SCRIPT>
<NOSCRIPT>
Hinweis auf fehlende Browserinterpretation
</NOSCRIPT>
```

Nun ist es zwar nicht so, dass jeder Browser, der kein JavaScript beherrscht, das **<NOSCRIPT>**-Tag korrekt interpretieren kann, aber dennoch ist dessen Einsatz sinnvoll. Denn gemäß einer alten Weisheit, nach der ein WWW-Browser für den Fall, dass er ein Tag nicht kennt, dessen Inhalt dennoch darstellt, ist die eigentliche Bedeutung des **<NOSCRIPT>**-Tags für den Browser nicht relevant. Warum trotz dieser Aussage statt jedes beliebigen dennoch das **<NOSCRIPT>**-Tag eingesetzt werden sollte? Die beiden großen Browser beherrschen diese Möglichkeit der Alternativdarstellung und zudem gehört das **<NOSCRIPT>**-Tag zum offiziellen HTML-4.0-Sprachstandard und wurde eben speziell für den Zweck einer alternativen Darstellungsweise entwickelt.

Die fehlende JavaScript-Fähigkeit oder die Einflussname von Anwendern auf diese einmal außer Acht gelassen, sollte man dennoch den JavaScript-Code vor solchen Browsern verstecken. Nur so kann gewährleistet werden, dass der Quellcode des JavaScripts dem Anwender nicht fälschlicherweise als Text präsentiert wird. Denn so gut die Eigenschaft der Browser, einen nicht bekannten Tag zu ignorieren, aber dennoch dessen Inhalt darzustellen, in vielerlei Hinsicht auch sein mag, bezüglich des **<SCRIPT>**-Tags ist diese Behandlung problematisch. Denn bei deren konsequenter Umsetzung würde dem Betrachter der reine JavaScript-Code angezeigt werden. Um dies zu umgehen, sollte der gesamte JavaScript-Block in einen HTML-Kommentar eingebettet werden. So wird in jedem Fall garantiert, dass diejenigen Clients, die das **<SCRIPT>**-Tag nicht interpretieren können, zumindest nicht dessen Inhalt auslesen.

```
<SCRIPT type="JavaScript">
<!--
Code vor normalem Browser verstecken
Inhalt des JavaScript-Programms
//-->
</SCRIPT>
```

Beachten Sie, dass es sich hierbei um eine Art der Auszeichnung handelt, die JavaScript-Code vor Browsern versteckt, denen die Bedeutung des **<SCRIPT>**-Tags nicht geläufig ist. Es handelt sich jedoch nicht um die Auszeichnung von Kommentaren in JavaScript bzw. deren Kennzeichnung. Weiterführende Informationen zu Kommentaren finden Sie im Abschnitt 3.1.

2.3 Fragen und Übungen

1. Mit welchem Attribut sollten JavaScript-Programme im einleitenden **<SCRIPT>**-Tag gekennzeichnet werden?

   language

   type

   src

2. Wie lautet der MIME-Typ von JavaScript?

   javascript

   javascript/text

   text/javascript

3. Nennen Sie die Aufgabe des **charset**-Attributs.

4. Welches der folgenden ISO-Zeichensätze ist der für den westeuropäischen Sprachraum geeignetste?

   ISO-8859-1

   ISO-8859-2

   ISO-8859-3

   ISO-8859-4

5. Warum führt die folgende Syntax zu keiner Anzeige im Browser, aber auch zu keiner Fehlermeldung?

```
<SCRIPT type="JavaScript">
<!--
document.write("Hallo Welt")>
//-->
</SCRIPT>
```

3 Syntax

Ziele dieses Kapitels

In diesem Kapitel lernen Sie die grundlegende Syntax von JavaScript kennen. Die vermittelten Informationen sind für die JavaScript-Programmierung unerlässlich. Nur wenn diese Grundlagen verstanden wurden, lassen sich erste Anwendungen erstellen. Besonderes Augenmerk ist auf den Abschnitt 3.3 zu richten. Hier werden die notwendigen Konventionen bezüglich der Namensvergabe gezeigt.

3.1 Kommentare

übersichtliche Programme

Durch Kommentare lässt sich die Lesbarkeit von JavaScript-Programmcode verbessern. Zusätzlich eignet sich der Einsatz von Kommentaren, um ein späteres Ändern der Syntax effizient zu gestalten und somit schnell auf die programmspezifischen Besonderheiten und deren Anforderungen reagieren zu können. Innerhalb der JavaScript-Spezifikation wurden zwei grundsätzliche Arten von Kommentar-Schreibweisen festgelegt:

- Zwei Schrägstriche kennzeichnen einen Kommentar bis zum Zeilenende.
- Durch die Zeichenabfolge /* und */ wird ein Blockkommentar, also ein Kommentar über mehrere Zeilen, spezifiziert.

Die zweite Kommentarform kann neben einem Blockkommentar auch einen einzeiligen Bereich kennzeichnen. Dies macht deutlich, dass für die Vereinheitlichung und eine verbesserte Überschaubarkeit des JavaScript-Quellcodes sehr wohl auf diese eine Art der Kommentar-Kennzeichnung zurückgegriffen werden kann und sollte. Es ergibt sich hieraus ein weitaus homogeneres Schriftbild, als dies durch die Zuweisung zweier unterschiedlicher Kommentarvarianten erzielt werden würde.

```
<SCRIPT type="text/JavaScript">
<!--
function goto(n)
{
this.length = n
return this
}
var urls = new goto(3)
/* es folgt die Definition
mehrerer URLs */
urls[0] = ""//leerer URL
urls[1] = "http://www.kss.de" //erster URL
urls[2] = "http://www.phoenix.de" //zweiter URL
function begin(url){
if (url.control.desert>0)
location.href = urls[url.control.desert]
}
//-->
</SCRIPT>
```

In dem Beispiel wurden beide Varianten der Kommentarauszeichnung angewandt. Neben den bereits erwähnten Einsatzmöglichkeiten eignen sich Kommentare zusätzlich für die Unterbringung von Urheberrechtsbestimmungen für das Script und für Änderungsanleitungen.

3.2 Anweisungen

Folge von Anweisungen

Ohne bereits an dieser Stelle ausführlich auf die zahlreichen Notationsvarianten von JavaScript einzugehen, kann dennoch die Aussage getroffen werden, dass es sich bei dieser Programmiersprache letztendlich um nichts anderes als eine Anordnung von Anweisungen handelt. Bei diesen Anweisungen handelt es sich um Befehle, die der WWW-Browser interpretiert und anschließend in auf dem System des Nutzers ausführbaren Maschinencode umwandelt. Es existieren freilich die verschiedensten Formen von Anweisungen wie zum Beispiel das Zuweisen von Werten zu einer Variablen. Aber auch die folgende Syntax stellt eine Anweisung dar:

```
alert("Corporate Identitiy");
```

Anweisungen sollten stets durch einen Strichpunkt abgeschlossen werden. Zwar ist dies nicht in jedem Fall notwendig, um eventuelle Fehlermeldungen bereits im Vorfeld einzudämmen, sollte dennoch jede Anweisung hierdurch beendet werden.

3.2.1 Anweisungsblöcke

geschweifte Klammern

Ein Anweisungsblock setzt sich aus mindestens zwei Anweisungen zusammen, die sich innerhalb einer übergeordneten Anweisung oder in Gänze innerhalb einer Funktion befinden. Das Charakteristische an Anweisungsblöcken ist das notorische Vorhandensein von geschweiften Klammern. Im Zusammenhang mit Funktionen ist zu sagen, dass deren Anfang und Ende jeweils durch eine geschweifte Klammer gekennzeichnet sind und es sich bei Funktionen somit um Anweisungsblöcke handelt. Ein Beispiel, welches einen Auszug einer Funktion zeigt, soll die für diese Notationsvariante typische Syntax demonstrieren.

```
function goto(n)
{
this.length = n
return this
}
```

Verschachtelungen

In JavaScript ist die Verschachtelung von Anweisungsblöcken erlaubt. Bei einer solchen Verschachtelung ist darauf zu achten, dass einer öffnenden geschweiften Klammer jeweils eine schließende geschweifte Klammer zugewiesen werden muss. Aus diesem Grund ist es sinnvoll, jede geschweifte Klammer innerhalb einer separaten Zeile zu notieren, um die Übersichtlichkeit, und dies gilt insbesondere im Zusammenhang mit komplexen JavaScript-Programmen, zu wahren.

3.3 Namensvergabe

Regeln für Namen

Während der Programmierung eines JavaScript-Programms müssen häufig selbst kreierte Namen vergeben werden. Dies kann beispielsweise bei eigenen Funktionen, Variabelendeklarationen oder im Zusammenhang mit eigenen Objekten der Fall sein. Diese Namen unterliegen gewissen Konventionen, ohne deren vollständige Berücksichtigung unweigerlich Fehlermeldungen des verwendeten Browsers ausgegeben werden. Die grundlegendsten und in jedem Fall zu berücksichtigenden Regeln sind hier in einer Übersicht zusammengefasst.

- Namen dürfen keine deutschen Umlaute enthalten.
- Namen dürfen keine Leerzeichen enthalten.
- Als einziges gestattetes Sonderzeichen ist der Unterstrich vorgesehen.
- Es dürfen keine reservierten Namen verwendet werden. (Mehr zu diesem Thema finden Sie im Abschnitt 3.3.1).
- Namen sollten prinzipiell aus nicht mehr als 32 Zeichen bestehen.
- Es wird zwischen Groß- und Kleinschreibung unterschieden.

- Namen sollten stets mit einem Buchstaben beginnen.
- Namen dürfen sowohl aus Buchstaben wie auch aus Zahlen oder einer Symbiose aus beiden Elementen bestehen.

Problematisch wird die Vergabe von Namen immer dann, wenn sich deren Bedeutung nicht auf den ersten Blick erschließt. Zwar ist es gang und gäbe, kryptische Namen einzusetzen, vor dieser Verfahrensweise sei an dieser Stelle dennoch gewarnt. So mag zwar ein bestimmter Name bzw. dessen Bedeutung während der ersten Programmierzeit noch geläufig sein, bei später anfallenden Korrekturen des Internetprojekts ist das dann allerdings häufig nicht mehr der Fall. Erschwerend kommt hinzu, dass an der Erstellung eines komplexen Projekts häufig mehrere Programmierer beteiligt sind. Und eben auch für die sollten die Bedeutung und auch die Schreibweise eines Namens keine zusätzliche Belastung darstellen.

3.3.1 Reservierte Namen

diese Namen nicht verwenden

Die folgende Tabelle enthält alle reservierten Namen, die innerhalb eines JavaScript-Programms nicht für selbst definierte Variablen oder Objekte verwendet werden dürfen. Wie die Auflistung zeigt, ist die Mehrzahl dieser Namen derzeit noch mit keiner Funktion belegt. Sie sollen aber für spätere Sprachweiterentwicklungen reserviert bleiben. Es ist in jedem Fall darauf zu achten, dass auch solche Namen, die noch nicht mit einer Funktion belegt sind, nicht als Variablen- oder Funktionsnamen verwendet werden dürfen.

Wort	Beschreibung
`abstract`	bislang nicht belegt
`boolean`	bislang nicht belegt
`break`	beendet eine Schleife
`byte`	bislang nicht belegt
`case`	einzelner Fall innerhalb einer Fallunterscheidung
`catch`	Fehlerbehandlung
`char`	bislang nicht belegt
`class`	bislang nicht belegt
`const`	Deklaration von Konstanten
`continue`	erzwingt den nächsten Schleifen-Durchlauf
`debugger`	bislang nicht belegt
`default`	wenn kein Fall bei einer Fallunterscheidung zutrifft
`delete`	löscht ein Objekt

Tabelle 3.1: Reservierte Namen in JavaScript-Programmen

Wort	Beschreibung
`do`	für `do-while`-Schleifen
`double`	bislang nicht belegt
`else`	Sonst-Fall bei Fallunterscheidungen
`enum`	bislang nicht belegt
`export`	Objekte für fremde Scripts ausführbar machen
`extends`	bislang nicht belegt
`false`	Rückgabewert von Funktionen
`final`	bislang nicht belegt
`finally`	bislang nicht belegt
`float`	bislang nicht belegt
`for`	für for-Schleife
`function`	Funktionsdeklaration
`goto`	bislang nicht belegt
`if`	Bedingung für bedingte Anweisungen
`implements`	bislang nicht belegt
`import`	erlaubt das Importieren von Objekten usw. aus signierten Scripts
`in`	für `for-in`-Schleifen
`instanceof`	überprüft, ob eine Variable eine Instanz eines Objekts ist
`int`	bislang nicht belegt
`interface`	bislang nicht belegt
`long`	bislang nicht belegt
`native`	bislang nicht belegt
`new`	definiert ein Objekt
`null`	setzt ein Objekt auf null
`package`	bislang nicht belegt
`private`	bislang nicht belegt
`protected`	bislang nicht belegt
`public`	bislang nicht belegt
`return`	Rückgabewert für Funktionen
`short`	bislang nicht belegt
`static`	bislang nicht belegt
`super`	bislang nicht belegt
`switch`	Fallunterscheidung
`synchronized`	bislang nicht belegt
`this`	nimmt auf das aktuelle Objekt Bezug
`throw`	nutzerdefinierte Ausnahme
`throws`	bislang nicht belegt

Tabelle 3.1: Reservierte Namen in JavaScript-Programmen (Forts.)

Wort	Beschreibung
`transient`	bislang nicht belegt
`true`	Rückgabewert von Funktionen
`try`	testet, ob eine Anweisung ausführbar ist
`typeof`	ermittelt den Typ eines Elements
`var`	Variabelendeklaration
`void`	erzeugt keinen Rückgabewert
`volatile`	bislang nicht belegt
`while`	für `while`-Schleifen
`with`	einem Objekt werden mehrere Anweisungen zugewiesen

Tabelle 3.1: Reservierte Namen in JavaScript-Programmen (Forts.)

Diese Tabelle sollte bei der Programmierung eines jeden JavaScripts berücksichtigt werden, da auch die Verwendung von derzeit noch nicht belegten Namen zu einer JavaScript-Fehlermeldung führen kann. Das soll Abbildung 3.1 demonstrieren. Zu der Fehlermeldung kam es, da eine Funktion mit dem Namen `long()` definiert wurde und `long` ein reservierter Name ist.

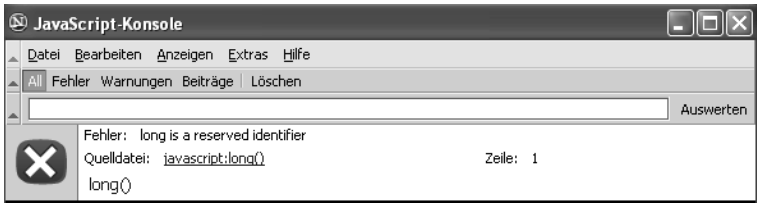


Abbildung 3.1: Fehlermeldung im Netscape Navigator 7

3.4 Fragen und Übungen

1. Welche der folgenden selbst vergebenen Namen sind korrekt?

   bär

   1bär

   Fotos2

   final

   neues:Foto

2. Woran erkennen Sie einen Anweisungsblock?

4 Sprachelemente

lernen

Ziele dieses Kapitels

In diesem Kapitel lernen Sie die Sprachelemente von JavaScript kennen. Neben dem Umgang mit Datentypen erfahren Sie, wie sich Variablen definieren und deklarieren lassen. Um effektiv mit JavaScript arbeiten zu können, benötigen Sie Wissen darüber, wie sich Schleifen und bedingte Abfragen realisieren lassen. Beachten Sie, dass es sich bei den hier vorgestellten Sprachelementen um grundlegende Programmierkenntnisse handelt. Haben Sie das Prinzip von Schleifen usw. in JavaScript verstanden, lässt sich dieses Wissen auch auf andere Programmiersprachen anwenden. Weitere Eckpfeiler dieses Kapitels sind Operatoren und die Ereignisbehandlung.

4.1 Datentypen

Daten manipulieren

In JavaScript-Programmen findet im Regelfall eine Manipulation von Daten statt. Durch die Verwendung von Datentypen kann bekannt gegeben werden, in welcher Form die Daten vorkommen und welche Operationen auf ihnen möglich sind. JavaScript kennt die folgenden Datentypen und kann diese korrekt interpretieren.

- Boolesche Werte
- Strings
- Zahlen

Diese Datentypen stehen in unterschiedlichen Formen zur Verfügung. So können diese als Objekte, aber auch als einfache Werte bereitstehen.

Boolesche Werte

Wahrheitswerte

Der Datentyp der booleschen Werte fasst die beiden Wahrheitswerte `true` (wahr) und `false` (falsch) zusammen. Eingesetzt werden boolesche Datentypen vor allem für das Vergleichen von Werten. Auf diese Weise kann ein eindeutiges Ergebnis ermittelt werden. So kann beispielsweise

der Vergleich „9 größer 3?“ eindeutig den Wert **true** zurückliefern, da ja die Ziffer 9 tatsächlich größer als 3 ist. An den entsprechenden Stellen dieses Buches wird spezifisch auf boolesche Werte eingegangen, handelt es sich doch um einen in jedem Fall zu beherrschenden JavaScript-Datentyp.

Strings

Zeichenketten

Unter dem Oberbegriff Strings wird eine beliebige Abfolge von Zeichen zusammengefasst, woraus sich auch das deutsche Pendant des Namens ergibt: Zeichenketten. Innerhalb einer Zeichenkette können alle Zeichen, die auf einer Computertastatur zu finden sind, eingesetzt werden. Neben Buchstaben, Ziffern usw. kennt JavaScript einige Sonderzeichen, die vor allem im Hinblick auf eine komfortablere Textausgabe sinnvoll sind. Einen Überblick aller in JavaScript spezifizierten Sonderzeichen liefert die nachstehende Tabelle. Beachten Sie, dass alle Sonderzeichen mit einem vorangestellten Backslash versehen werden müssen.

Sonderzeichen	Bedeutung
`\\`	Backslash
`\'`	Single Quote
`\"`	Quote
`\b`	Backspace
`\f`	Form Feed
`\n`	New Line
`\t`	Tabulator
`\r`	Carriage Return

Tabelle 4.1: Sonderzeichen in JavaScript

zulässige Länge von Strings

Grundsätzlich spielt die Länge von Strings für deren korrekte Darstellbarkeit keine Rolle. Dennoch sollte, und dies gilt insbesondere im Hinblick auf den Netscape Navigator, eine Länge von 63 KByte nicht überschritten werden. Um eine korrekte Darstellung zu erzielen, sollten Strings, deren Länge über 256 Zeichen hinausgeht, in Teilstrings aufgesplittet werden. Strings werden durch Anführungszeichen gekennzeichnet. Es sind sowohl einfache wie auch doppelte Anführungszeichen gestattet. Das folgende Beispiel zeigt einen korrekt definierten String, der in einfache Anführungszeichen gesetzt ist und zusätzlich doppelte zur gesonderten Auszeichnung des Begriffs `String` enthält.

```
'Ein simpler "String", also eine Zeichenkette'
```

Da eine solche Schreibweise im täglichen Gebrauch tatsächlich vermehrt auftritt, sei an dieser Stelle ausdrücklich auf die korrekte Handhabung von Anführungszeichen hingewiesen. Es ist auf ein hierarchisches

Vorhandensein von Anführungszeichen zu achten. Dies bedeutet, dass jeder String exakt mit den Anführungszeichen beendet werden muss, mit denen er auch eingeleitet wurde. Exemplarisch für eine fehlerhafte String-Auszeichnung ist die folgende Syntax.

```
"Ein simpler "String", also eine Zeichenkette"
```

Zwar ist korrekterweise für jedes öffnende auch ein schließendes Anführungszeichen vorhanden, der JavaScript-Interpreter kann diesen String dennoch nicht fehlerfrei darstellen, würden sich doch durch das gezeigte Vorgehen die zu Beginn gesetzten doppelten Anführungszeichen noch vor dem Ende des Strings wieder aufheben.

Zahlen

keine Unterscheidung zwischen Real- und Integer-Zahlen

Die Notation von Zahlen spielt in JavaScript eine wichtige Rolle. So lassen sich Zahlen neben dem Einsatz für einfache oder komplexe Berechnungen beispielsweise auch für die Definition von Farbwerten verwenden. JavaScript birgt in der Behandlung von Zahlen gegenüber anderen Programmiersprachen den Vorteil in sich, dass hier keine explizite Unterscheidung zwischen ganzen und Zahlen mit einem Nachkommaanteil unternommen wird. In JavaScript werden stattdessen ganze Zahlen als 32-Bit-Zahlen und Fließkommazahlen als 64-Bit-IEEE-Zahlen gespeichert. Während die Notation von ganzen Zahlen keine gesonderten Anforderungen und Syntaxvariationen verlangt, sind in JavaScript diverse Zahlenvarianten möglich, die speziell ausgezeichnet werden müssen und deren Verständnis zumindest ein gewisses Maß an mathematischem Grundwissen voraussetzt. Die folgende Auflistung enthält alle in JavaScript zulässigen Zahlen bzw. Zahlenbereiche.

- 8 Byte lange: Deren Gültigkeitsbereich erstreckt sich von +/-2.2250738585072014*10-308 bis +/-1.7976931348623157*10308.
- Dezimal: Hierzu gehören natürliche und rationale Zahlen sowie Zahlen in Expotentialschreibweise.
- Hexadezimal: Hierzu gehören Ganzzahlen mit einem führenden 0X beziehungsweise einem führenden 0x.
- Oktal: Dies sind Ganzzahlen mit einer führenden Null.

Der Datentyp **`number`** kennt weitere mögliche Werte, die für spezielle Anwendungen bereitstehen, auf deren Einsatz im Laufe dieses Buches noch ausführlich eingegangen werden wird. Im Einzelnen sind dies:

- **`Infinity`** – Unendliche Zahl als größte Darstellbare (**>**).
- **`-Infinity`** – Unendlich kleine Zahl als kleinste Darstellbare (**<**).
- **`NaN`** – Beschreibt ein undefiniertes Ergebnis einer Rechnung (**`NanN`** = Not a Number).

Für nicht-ganzzahlige Zahlen gilt, dass die Trennung zwischen ganzzahligem und gebrochenem Anteil nicht durch ein Komma, sondern durch einen Punkt gekennzeichnet wird. Bei der Verwendung von hexadezimalen Zahlen ist zudem zu beachten, dass zwischen Groß- und Kleinschreibung unterschieden wird.

Umrechnungstabelle

hexadezimale, dezimale und oktale Zahlen

Neben der einfachsten Notationsform von Zahlen, nämlich der dezimalen Schreibweise, können Zahlenwerte in JavaScript auch in oktaler- und hexadezimaler Form verwendet werden. Um diese alternativen Zahlenwerte einsetzen zu können, müssen gesonderte Schreibweisen verwendet werden. Hexadezimalzahlen wird ein 0x bzw. 0X vorangestellt. Oktale Zahlen werden durch eine vorangestellte 0 gekennzeichnet. Ein weiteres Unterscheidungskriterium betrifft die jeweiligen Zahlenbereiche. Bei oktalen Zahlen kann der Wertebereich zwischen 0 und 7 verwendet werden. Bei hexadezimalen Werten ist indes ein Wertebereich zwischen 0 und 9 sowie A bis F bzw. a bis f gestattet. Folgende Tabelle beinhaltet alle Umrechnungen für Zahlen zwischen 0 und 255.

Dez	Oct	Hex	Dez	Oct	Hex	Dez	Oct	Hex
0	00	0X0	86	0126	0X56	172	0254	0XAC
1	01	0X1	87	0127	0X57	173	0255	0XAD
2	02	0X2	88	0130	0X58	174	0256	0XAE
3	03	0X3	89	0131	0X59	175	0257	0XAF
4	04	0X4	90	0132	0X5A	176	0260	0XB0
5	05	0X5	91	0133	0X5B	177	0261	0XB1
6	06	0X6	92	0134	0X5C	178	0262	0XB2
7	07	0X7	93	0135	0X5D	179	0263	0XB3
8	010	0X8	94	0136	0X5E	180	0264	0XB4
9	011	0X9	95	0137	0X5F	181	0265	0XB5
10	012	0XA	96	0140	0X60	182	0266	0XB6
11	013	0XB	97	0141	0X61	183	0267	0XB7
12	104	0XC	98	0142	0X62	184	0270	0XB8
13	015	0XD	99	0143	0X63	185	0271	0XB9
14	106	0XE	100	0144	0X64	186	0272	0XBA
15	017	X0F	101	0145	0X65	187	0273	0XBB
16	020	0X10	102	0146	0X66	188	0274	0XBC
17	021	0X11	103	0147	0X67	189	0275	0XBD
18	022	0X12	104	0150	0X68	190	0276	0XBE
19	023	0X13	105	0151	0X69	191	0277	0XBF

Tabelle 4.2: Umrechnungstabelle der Zahlen 0 bis 255

Dez	Oct	Hex	Dez	Oct	Hex	Dez	Oct	Hex
20	024	0X14	106	0152	0X6A	192	0300	0XC0
21	025	0X15	107	0153	0X6B	193	0301	0XC1
22	026	0X16	108	0154	0X6C	194	0302	0XC2
23	027	0X17	109	0155	0X6D	195	0303	0XC3
24	030	0X18	110	0156	0X6E	196	0304	0XC4
25	031	0X19	111	0157	0X6F	197	0305	0XC5
26	032	0X1A	112	0160	0X70	198	0306	0XC6
27	033	0X1B	113	0161	0X71	199	0307	0XC7
28	034	0X1C	114	0162	0X72	200	0310	0XC8
29	035	0X1D	115	0163	0X73	201	0311	0XC9
30	036	0X1E	116	0164	0X74	202	0312	0XCA
31	037	0X1F	117	0165	0X75	203	0313	0XCB
32	040	0X20	118	0166	0X76	204	0314	0XCC
33	041	0X21	119	0167	0X77	205	0315	0XCD
34	042	0X22	120	0170	0X78	206	0316	0XCE
35	043	0X23	121	0171	0X79	207	0317	0XDF
36	044	0X24	122	0172	0X7A	208	0320	0XD0
37	045	0X25	123	0173	0X7B	209	0321	0XD1
38	046	0X26	124	0174	0X7C	210	0322	0XD2
39	047	0X27	125	0175	0X7D	211	0323	0XD3
40	050	0X28	126	0176	0X7E	212	0324	0XD4
41	051	0X29	127	0177	0X7F	213	0325	0XD5
42	052	0X2A	128	0200	0X80	214	0326	0XD6
43	053	0X2B	129	0201	0X81	215	0327	0XD7
44	054	0X2C	130	0202	0X82	216	0330	0XD8
45	055	0X2D	131	0203	0X83	217	0331	0XD9
46	056	0X2E	132	0204	0X84	218	0332	0XDA
47	057	0X2F	133	0205	0X85	219	0333	0XDB
48	060	0X30	134	0206	0X86	220	0334	0XDC
49	061	0X31	135	0207	0X87	221	0335	0XDD
50	062	0X32	136	0210	0X88	222	0336	0XDE
51	063	0X33	137	0211	0X89	223	0337	0XDF
52	064	0X34	138	0212	0X8A	224	0340	0XE0
53	065	0X35	139	0213	0X8B	225	0341	0XE1
54	066	0X36	140	0214	0X8C	226	0342	0XE2
55	067	0X37	141	0215	0X8D	227	0343	0XE3
56	070	0X38	142	0216	0X8E	228	0344	0XE4

Tabelle 4.2: Umrechnungstabelle der Zahlen 0 bis 255 (Forts.)

Dez	Oct	Hex	Dez	Oct	Hex	Dez	Oct	Hex
57	071	0X39	143	0217	0X8F	229	0345	0XE5
58	072	0X3A	144	0220	0X90	230	0346	0XE6
59	073	0X3B	145	0221	0X91	231	0347	0XE7
60	074	0X3C	146	0222	0X92	232	0350	0XE8
61	075	0X3D	147	0223	0X93	233	0351	0XE9
62	076	0X3E	148	0224	0X94	234	0352	0XEA
63	077	0X3F	149	0225	0X95	235	0353	0XEB
64	0100	0X40	150	0226	0X96	236	0354	0XEC
65	0101	0X41	151	0227	0X97	237	0355	0XED
66	0102	0X42	152	0230	0X98	238	0356	0XEE
67	0103	0X43	153	0231	0X99	239	0357	0XEF
68	0104	0X44	154	0232	0X9A	240	0360	0XF0
69	0105	0X45	155	0233	0X9B	241	0361	0XF1
70	0106	0X46	156	0234	0X9C	242	0362	0XF2
71	0107	0X47	157	0235	0X9D	243	0363	0XF3
72	0110	0X48	158	0236	0X9E	244	0364	0XF4
73	0111	0X49	159	0237	0X9F	245	0365	0XF5
74	0112	0X4A	160	0240	0XA0	246	0366	0XF6
75	0113	0X4B	161	0241	0XA1	247	0367	0XF7
76	0114	0X4C	162	0242	0XA2	248	0370	0XF8
77	0115	0X4D	163	0243	0XA3	249	0371	0XF9
78	0116	0X4E	164	0244	0XA4	250	0372	0XFA
79	0117	0X4F	165	0245	0XA5	251	0373	0XFB
80	0120	0X50	166	0246	0XA6	252	0374	0XFC
81	0121	0X51	167	0247	0XA7	253	0375	0XFD
82	0122	0X52	168	0250	0XA8	254	0376	0XFE
83	0123	0X53	169	0251	0XA9	255	0377	0XFF
84	0124	0X54	170	0252	0XAA			
85	0125	0X55	171	0253	0XAB			

Tabelle 4.2: Umrechnungstabelle der Zahlen 0 bis 255 (Forts.)

4.2 Variablen

veränderbare Ausdrücke

JavaScript unterscheidet zwischen veränderlichen und unveränderlichen Ausdrücken. Als unveränderbare werden die bereits vorgestellten Sprachelemente wie Zahlen, Strings usw. verstanden. In diesem Abschnitt soll auf veränderbare Elemente eingegangen werden. Diese, in einigen Fällen auch „Veränderliche“ genannten Variablen dienen dem Speichern von Werten, die sich während der Laufzeit des JavaScript-

Programms ändern. Es existieren grundlegende Regeln, die bei der Definition von Variablen zu beachten sind. Diese beschreiben beispielsweise eindeutig, welche Elemente innerhalb eines Variablennamens verwendet werden dürfen. Gleichzeitig ist aber auch festgeschrieben, welchen Gültigkeitsbereich eine Variable besitzen kann und muss.

4.2.1 Variablen deklarieren

globale und lokale Variablen

Für die Definition von Variablen existieren fest vorgeschriebene Regeln. So sind beispielsweise im Hinblick auf die Vergabe von Namen exakt festgeschriebene Notationsbesonderheiten zu beachten. Zudem wird in JavaScript zwischen globalen und lokalen Variablen unterschieden. Diese Unterscheidung ergibt sich aus dem Zusammenhang zwischen einzelnen Funktionen und dem Hauptprogramm. Denn eine Funktion ist jeweils ein Unterabschnitt des Hauptprogramms, der als Anweisungsblock an einer beliebigen Stelle des Hauptprogramms definiert wurde und von dem aus Programme oder andere Funktionen aufgerufen werden können. Der Gültigkeitsbereich einer globalen Variablen erstreckt sich hingegen über das gesamte Programm und kann somit an jeder Stelle aufgerufen und deren Wert überall verändert werden. Bei der Vergabe eines Variablennamens sind die folgenden Varianten möglich:

Kriterien für Variablennamen

- Namen als Variablen
- Numerische Variablen
- Strings
- Boolesche Variablen

kaum Typisierungen

Beachten Sie, dass JavaScript im Zusammenhang mit Variablennamen zwischen Groß- und Kleinschreibung unterscheidet. Zudem sind alle reservierten Wörter, und hierbei spielt es keine Rolle, ob diese Wörter bereits mit Funktionen belegt sind oder nicht, nicht einsetzbar und führen häufig zu einer JavaScript-Fehlermeldung. In JavaScript ist eine Typisierung von Variablen, wie beispielsweise in Pascal, offener ausgelegt. Eine Unterscheidung zwischen char, string, real und integer kann nur bedingt vorgenommen werden. Denn außer einer Unterscheidung zwischen numerischen und nicht-numerischen Variablen finden in JavaScript in dieser Hinsicht keine weiteren Typisierungen statt. So ist es freilich auch nicht verwunderlich, dass der Inhalt von numerischen Variablen ohne eine vorherige Konvertierung in einen String direkt dargestellt werden kann.

Es existieren verschiedene Syntaxformen, mit denen Variablen definiert werden können. Zwar ist es nicht zwingend vorgeschrieben, aus Gründen der besseren Lesbarkeit des Programmcodes sollte dennoch jede Variable mit dem JavaScript-Schlüsselwort `var` definiert werden. Jede Vari-

ablendeklarationen sollte zudem mit einem Strichpunkt abgeschlossen werden. Die folgenden Deklarationsvarianten stehen im Zusammenhang mit Variablen zur Verfügung. Welche für einen ganz bestimmten Zweck verwendet werden sollte, lässt sich allerdings nicht pauschalisieren.

```
Variablenname = Wert;
```

Bei dieser Syntaxform wird zunächst der Variablenname, gefolgt von einem Gleichheitszeichen und dem sich hieran anschließenden Variablenwert, notiert.

```
var Variablenname = Wert;
```

Um die Lesbarkeit von Programmen zu erhöhen, sollten Variablen durch das Schlüsselwort **var** gekennzeichnet werden. Hieran schließen sich der Name der Variablen, ein Gleichheitszeichen sowie der Wert der Variablen an.

```
var Variablenname = Wert,  Variablenname1 = Wert1, Variablenname = Wert2;
```

Um mehrere Variablen platzsparend zu deklarieren, steht diese Syntaxform zur Verfügung. Hier wird zunächst durch das Schlüsselwort **var** die Deklaration der Variablen eingeleitet. Hinter dem Gleichheitszeichen folgen die Variablennamen sowie deren Werte. Die jeweiligen Variablendeklarationen werden durch Kommata voneinander getrennt.

Die Wahl von Variablennamen ist von entscheidender Bedeutung. Obwohl diesem Thema ein eigener Abschnitt gewidmet wird, sollen an dieser Stelle drei grundlegende Regeln bereits genannt werden.

- Es sollten systematische und mit einer bestimmten Bedeutung belegte Namen vergeben werden.
- Variablen sollen explizit deklariert werden.
- Diese Deklarationen sollten sinnvoll und treffend auskommentiert werden.

Im folgenden Beispiel wird die Variable `Hallo` deklariert. Dieser wird als Wert die Zeichenkette `Hallo Welt` zugewiesen. Diese Zeichenkette wird anschließend mittels der **`alert()`**-Methode in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
</HEAD>
</BODY>
<SCRIPT type="text/JavaScript">
var Hallo = "Hallo Welt"
alert(Hallo);
```

```
</SCRIPT>
</BODY>
</HTML>
```

Listing 4.1: Ausgabe der Variablen `Hallo`

Beachten Sie, dass die Deklaration von Variablen den Programmfluss zwar nicht in jedem Fall beeinflusst, für die Lesbarkeit von Programmen sind sie jedoch ein entscheidendes Stilmittel. Auf den folgenden Seiten werden Sie die verschiedenen Facetten von Variablen kennen lernen.

4.2.2 Variablennamen benennen

logische Bezeichner

Selbst dokumentierende Programme sollten das Ziel jedes Programmierers sein. Dies gelingt jedoch nur, wenn das Programm stilistisch sauber programmiert ist. Besonders wichtig ist in diesem Zusammenhang die Wahl der Variablennamen. Werden diese geschickt gewählt, erschließt sich beim Lesen eines Programms bereits dessen Logik, ohne dass umfangreiche Kommentare verwendet werden müssen. (Dies besagt jedoch nicht, dass auf Kommentare verzichtet werden soll!) Zur Einführung und als Veranschaulichung gut und schlecht gewählter Variablennamen sollen die nachstehenden Variablendeklarationen dienen.

```
Keine_ahnung=10
weiss_nicht=5
keinen_schimmer=Keine_ahnung+weiss_nicht
```

Diese Deklaration entspricht nicht den geringsten stilistischen Anforderungen. Weder wurden hier gleiche Konventionen für die Namen verwendet noch sind diese selbst erklärend. Die Lesbarkeit ist hier nur bedingt gegeben. Zumal dann, wenn man sich ein umfangreiches Programm vorstellt, welches komplett in diesem Stil verfasst wurde. Den gleichen Zweck, stilistisch jedoch sauber, erfüllt die folgende Syntax:

```
var Summand1 = 10
var Summand2 = 5
var Summe = Summand1 + Summand2
```

Der Programmcode ist hier selbst erklärend. Die Kennzeichnung der Variablen durch das Schlüsselwort **`var`** macht deutlich, dass hier Variablendeklarationen vorgenommen werden. Die Variablennamen sind so gewählt, dass deren Verwendungszweck auf den ersten Blick ersichtlich ist.

Regeln für Variablennamen

Neben stilistischen Fragen existieren für die Namensvergabe auch fest vorgeschriebene Regeln. Diese müssen in jedem Fall eingehalten werden. Ansonsten können JavaScript-Programme nicht ordnungsgemäß arbeiten und der Programmfluss bricht im Regelfall ab.

- Es dürfen keine Leerzeichen vorhanden sein!
- Umlaute und das „ß“ sind nicht zulässig!
- Bis auf den Unterstrich dürfen keine Sonderzeichen vorkommen!
- Das erste Zeichen muss ein Buchstabe sein!
- Zahlen sind erlaubt. Ausgenommen hiervon ist das erste Zeichen.
- Es wird zwischen Groß- und Kleinschreibung unterschieden!

Selbstverständlich können an dieser Stelle nur grundlegende Hinweise für die Wahl von Variablennamen gegeben werden. Wie Sie dies letztendlich bei der Programmierung handhaben, bleibt selbstverständlich allein Ihnen überlassen. Dennoch bleibt zu sagen, dass nur stilistisch gute Programme schnell lesbar sind und somit auf effiziente Weise modifiziert werden können. Als weiterer wichtiger Punkt sei die Nichtverwendung von reservierten Wörtern für Variablennamen genannt. Bei reservierten Wörtern handelt es sich um solche, die ausschließlich für einen bestimmten Zweck entwickelt wurden.

4.2.3 Globale und lokale Variablen

Gültigkeitsbereiche von Variablen

In JavaScript wird zwischen lokalen und globalen Variablen unterschieden. Jede Variable, die innerhalb des Hauptprogramms deklariert wurde, wird als globale Variable bezeichnet. Solche Variablen, die innerhalb von Funktion deklariert wurden, werden als lokale Variablen bezeichnet. Lokale Variablen müssen in jedem Fall durch das Schlüsselwort **`var`** gekennzeichnet werden. Im Zusammenhang mit Variablen ist auf deren Gültigkeitsbereich zu achten. Dieser beschreibt, auf welchen Teil des Programms bzw. auf welche Funktion die Variable angewandt werden darf. Eine Beschreibung des Gültigkeitsbereichs von Variablen schließt sich an diesen Abschnitt an. Zunächst soll der Unterschied zwischen lokalen und globalen Variablen anhand zweier Beispiele verdeutlicht werden. Folgendes Szenario soll mittels beider Anwendungen realisiert werden. Durch Anklicken zweier Hyperlinks sollen zwei Funktionen aufgerufen werden. Diese sollen innerhalb eines Meldungsfensters den Titel der Datei anzeigen. Zu diesem Zweck wird die **`title`**-Eigenschaft des **`document`**-Objekts verwendet. Diese Eigenschaft wird innerhalb der Variablen `Ausgabe` gespeichert und mittels der **`alert()`**-Methode ausgegeben.

```
<HTML>
<HEAD>
<title>Dokument-Titel</title>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
```

```
{
var Ausgabe = document.title;
alert(Ausgabe);
}
function zeigeWert2()
{
alert(Ausgabe);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:zeigeWert()">Funktion lokal</A>
<BR>
<A href="javascript:zeigeWert2()">Funktion lokal</A>
</BODY>
</HTML>
```

Listing 4.2: Veranschaulichung lokaler und globaler Variablen

Innerhalb der Funktion `zeigeWert()`wurde die Variable `Ausgabe` deklariert. Bezug nehmend auf das bereits vermittelte Wissen können wir sagen, dass es sich somit um eine lokale Variable handelt, die demzufolge auch lediglich innerhalb dieser Funktion verwendet werden kann. Zwar wird auch in der `zeigeWert2()`-Funktion auf diese Variable zugegriffen, dieser Versuch bleibt jedoch erfolglos, da der Gültigkeitsbereich der Variablen auf die erste Funktion beschränkt ist.

Wie beide Funktionen auf die `Ausgabe`-Variable zugreifen und den korrekten Wert ausgeben können, veranschaulicht die nachstehende Syntax. Zwar ist diese mit der vorangegangenen fast identisch, statt einer lokalen wird hier allerdings eine globale Variable verwendet.

```
<HTML>
<HEAD>
<TITLE>Dokument-Titel</TITLE>
<SCRIPT type="text/JavaScript">
<!--
var Ausgabe = document.title;
function zeigeWert()
{
alert(Ausgabe);
}
function zeigeWert2()
{
alert(Ausgabe);
}
```

```
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:zeigeWert()">Funktion global</A>
<BR>
<A href="javascript:zeigeWert2()">Funktion global</A>
</BODY>
</HTML>
```

Listing 4.3: Veranschaulichung lokaler und globaler Variablen

Die Variable `Ausgabe` wurde hier nicht innerhalb einer Funktion, sondern im Hauptprogramm notiert. Hieraus ergibt sich, dass beide Funktionen auf diese Variable zugreifen können und somit bei beiden Funktionsaufrufen der Seitentitel des Dokuments in einem Meldungsfenster angezeigt wird.

4.2.4 Gültigkeitsbereiche

Probleme beim Programmieren

In welchem Bereich Variablen gültig sind, ist für die Programmierung von entscheidender Bedeutung. Nur bei dem Verständnis dieses Sachverhalts lassen sich Probleme bei der Entwicklung von JavaScript-Programmen vermeiden. Aus diesem Grund wird hier anhand eines Beispiels vorgestellt, welchen Gültigkeitsbereich eine Variable besitzen kann. Variablen, die innerhalb des Hauptprogramms definiert sind, sind auch innerhalb von Funktionen gültig. Im Gegensatz hierzu sind Variablen, die innerhalb von Funktionen definiert sind, im Hauptprogramm nicht verwertbar. Das folgende Beispiel soll Ihnen die Gültigkeitsbereiche von lokalen und globalen Variablen veranschaulichen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var j = 10
alert("global " + j)
function wertEins()
{
alert("lokal_eins " + j)
j = 20
alert("lokal_eins " + j)
}
alert("global " + j)
function wertZwei()
{
alert("lokal_zwei " + j)
j = 30
```

```
alert("lokal_zwei " + j)
}
alert("global " + j)
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:wertEins()">Funktion wertEins</A>
<BR>
<A href="javascript:wertZwei()">Funktion wertZwei</A>
</BODY>
</HTML>
```

Listing 4.4: Gültigkeitsbereiche von Variablen

Als globale Variable wird hier `j` deklariert. Dieser wird als Wert `10` zugewiesen. Das Beispiel enthält zwei Funktionen. Diese bedienen sich zwar ebenfalls der Variablen `j`, weisen dieser aber jeweils andere Werte zu. Wie sich dieses Verhalten auf den Wert der Variablen `j` auswirkt, beschreibt die folgende Tabelle. Bei dieser handelt es sich um einen Programmablaufplan. In der linken Spalte finden Sie die Anweisungen, die im Programm notiert sind. Die mittlere Spalte beschreibt diese Anweisungen. In der rechten Spalte sind die Auswirkungen der entsprechenden Anweisungen zu sehen.

Anweisung	Beschreibung	Ausgabe
`var j = 10`	Definition und Initialisierung der Variablen `j` mit dem Wert `10`	keine
`alert("global "+ j)`	Ausgabe der globalen Variablen `j`	global 10
`function wertEins()`	Definition der Funktion `wertEins()`	keine
`alert("lokal_eins"+ j)`	Ausgabe der globalen Variablen `j`	lokal_eins 10
`j = 20`	Ändern der globalen Variablen `j`	keine
`alert("lokal_eins"+ j)`	Ausgabe der lokalen Variablen `j`	lokal_eins 20
`alert("global_eins"+j)`	Ausgabe der globalen Variablen `j`	global 10
`function wertZwei()`	Definition der Funktion `wertZwei()`	keine
`alert("lokal_zwei"+j)`	Ausgabe der lokalen Variablen `j`	lokal_zwei 20
`j=30`	Ändern der globalen Variablen `j`	keine
`alert("lokal_zwei"+ j)`	Ausgabe der lokalen Variablen `j`	lokal_zwei 30
`alert("global " + j)`	Ausgabe der globalen Variablen `j`	global 10

Tabelle 4.3: Veranschaulichung des Programmflusses und der Gültigkeitsbereiche

Wie Sie der Tabelle entnehmen können, bleibt der Wert der globalen Variablen `j` stets `10`. Dies gilt, obwohl dieser Variablen innerhalb der beiden Funktionen jeweils andere Werte zugewiesen wurden.

4.2.5 Konstanten definieren

bislang nur im Navigator 6 einsetzbar

Beachten Sie, dass die Möglichkeit der Definition von Konstanten erst mit der JavaScript-Version 1.5 eingeführt wurde. Zudem können Konstanten bislang lediglich vom Netscape Navigator ab Version 6 interpretiert werden. Im Internet Explorer erzeugen Konstanten eine Fehlermeldung. Konstanten werden in JavaScript durch das Schlüsselwort **const** gekennzeichnet. Bei Konstanten handelt es sich um das Gegenstück zu Variablen. Variablen dürfen innerhalb eines Programms ihren Wert verändern. Konstanten können dies nicht. Denn wie der Begriff schon sagt, ist der Wert einer Konstanten während des Programmablaufs konstant. Konstanten eignen sich demnach vor allem für solche Elemente, die in einem Programm häufig verwendet werden. Als Beispiel sei hier ein mathematisches JavaScript-Programm genannt, in dem permanent die Zahl PI eingesetzt wird. Da hier die Fehleranfälligkeit bei der Eingabe von PI enorm wäre, empfiehlt sich die Verwendung einer Konstanten. Ein Beispiel:

```
<HTML>
<HEAD>
</HEAD>
</BODY>
<SCRIPT type="text/JavaScript">
<!--
const PI = "3.141592654";
document.write(PI);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 4.5: Definition der Konstanten PI

Hinter dem Schlüsselwort **const** folgt der Konstantenname. Beachten Sie, dass hierbei die gleichen Regeln wie bei Variablennamen gelten. Um einer Konstanten einen Wert zuzuweisen, wird dieser hinter dem Gleichheitszeichen notiert.

4.3 Operatoren

Durch eine Operation werden im Regelfall zwei Literale oder Varibalen kombiniert. Hieraus wird ein Ergebniswert ermittelt. Operatoren sind ein unverzichtbares Syntaxelement, um Programme erstellen zu können. So lassen sich hierdurch nicht nur mathematische Ausdrücke berechnen, sondern auch Werte miteinander vergleichen. JavaScript kennt folgende Operatoren:

Arten von Operatoren

- arithmetische Operatoren
- logische Operatoren
- bitweise Operatoren
- Zuweisungsoperatoren
- Vergleichsoperatoren
- Stringoperatoren

Es existieren verschiedene Varianten, um Operatoren in JavaScript zu verwenden. Jeder Operator arbeitet mit einem oder zwei Operanden. Diese Unterscheidung spiegeln die beiden Begriffe unär (für Operatoren mit einem Operanden) und binär (für Operatoren mit zwei Operanden) wider.

```
Operator Operand
```

Bsp: `-9`

```
Operand Operator
```

Bsp: `i++`

```
Operand1 Operator Operand2
```

Bsp: `5 + 9`

Die nachstehende Deklaration von Variablen soll die Verwendung von Operatoren veranschaulichen. Innerhalb dieses Code-Fragments werden die vier Variablen `a`,`b`,`c` und `d` deklariert. Den Variablen `c` und `d` wird zu Beginn der Wert `5` zugewiesen. Der Ausdruck **++**`c`, welcher der Variablen `a` zugewiesen wird, bedeutet, dass der Wert der Variablen `c` um eins erhöht und anschließend der Variablen `a` zugewiesen wird. Somit ergibt sich für die Variablen `a` und `c` jeweils der Wert `4`. Der Variablen `b` wird der Ausdruck `d`**++** zugewiesen. Dieser legt fest, dass der Wert von `d` um eins, also auf `6`, erhöht wird. Der Wert der Variablen `b` ist `5`.

```
var a, b, c, d;
c = 5;
d = 5;
a=++c;
b=d++;
```

Beachten Sie, dass die verwendeten Operatoren und Operanden sich sinnvollerweise miteinander kombinieren lassen sollten. So macht zwar die Division zweier numerischer Werte sehr wohl Sinn, bei der Division zweier Zeichenketten aber nicht.

4.3.1 Arithmetische Operatoren

Schleifen und Zähler

Um mit Zahlen zu arbeiten, werden arithmetische Operatoren verwendet. Als Rückgabewerte werden Zahlen geliefert. Arithmetische Operatoren können jedoch mehr leisten als das Addieren oder Subtrahieren. So lässt sich der Operator **++** innerhalb von Schleifen beispielsweise dazu verwenden, um einen Wert um jeweils eins zu erhöhen.

Operator	Beschreibung	Beispiel	Wert
+	Addition	A=5+2	A=7
-	Subtraktion	A=5-2	A=3
/	Division	A=5/2	A=2.5
*****	Multiplikation	A=5*2	A=10
%	Modulo	A=5%2	A=1
++	Inkrement	A=3++	A=3+1
--	Dekrement	A=3--	A=3-1

Tabelle 4.4: Arithmetische Operatoren

Punktrechnung geht vor Strichrechnung

Beachten Sie, dass die hier vorgestellten Operatoren ausschließlich auf Zahlen angewandt werden dürfen. Sie können mehrere arithmetische Operatoren in Reihe verwenden. Hier gilt dann die übliche mathematischen Regel: Punktrechnung vor Strichrechnung. Soll diese Regel nicht angewandt werden, müssen die entsprechenden Ausdrücke in Klammern gesetzt werden.

Abkürzungen für arithmetische Operatoren

Nachteile von Abkürzungen

Die Schreibweise arithmetischer Operatoren lässt sich abkürzen. Sie sollten hierbei jedoch bedenken, dass diese Abkürzungen weitaus weniger verbreitet sind als die ausführliche Schreibweise. Hier kann es demnach zu Konflikten mit anderen JavaScript-Entwicklern kommen, denen diese Ausdrücke nicht geläufig sind. Dies macht eine Nachbearbeitung des Programms für andere Entwickler freilich ungleich schwerer.

Operator	Bedeutung	Langform	Kurzform
+=	Addition	Wert = Wert+5	Wert+=5
-=	Subtraktion	Wert = Wert-6	Wert-=6
/=	Division	Wert = Wet/7	Wert/=7
*=	Multiplikation	Wert = Wet*9	Wert*=9
%=	Modulo	Wert = Wert%2	Wert%=2

Tabelle 4.5: Abkürzungen arithmetischer Operatoren

Trotz der genannten „Nachteile“ dieser Abkürzungen bieten diese enorme Möglichkeiten, um JavaScript-Quellcode zu minimieren. Einer Verwendung von Abkürzungen für arithmetische Operatoren kann an dieser Stelle also nichts entgegengesetzt werden.

4.3.2 Logische Operatoren

Wahrheitswerte

Der Begriff der logischen Operatoren stammt aus der Algebra und der hierin verankerten Aussagenlogik, die auf Boole basiert. Logikoperatoren verknüpfen die booleschen Werte **`false`** und **`true`** und liefern als Ergebnis einen ebensolchen Wert zurück. Anhand dieser Aussage wird deutlich, dass hier stets eine absolute Fallunterscheidung zwischen einer wahren und einer falschen Aussage stattfindet. Ein Ausdruck „vielleicht“ existiert nicht.

Operator	Beschreibung	Beispiel	Wert
`&&`	Logisches UND	`2<3 && 3<5`	**`true`**
`||`	Logisches ODER	`5<6 || 6<4`	**`true`**
`!`	Logisches NICHT	**`!false`**	**`true`**

Tabelle 4.6: Logische Operatoren

Die aufgeführten Operatoren werden hauptsächlich im Zusammenhang mit bedingten Anweisungen und Schleifen eingesetzt. So lassen sich durch den Operator **`&&`** zwei Ausdrücke miteinander verknüpfen und lässt sich deren Wahrheitsgehalt überprüfen. Nur wenn beide Ausdrücke der Bedingung erfüllt sind, kann der Anweisungsblock ausgeführt werden.

4.3.3 Bitweise Operatoren

Bei bitweisen Operatoren werden die Operanden nicht als Zahlen, sondern als binäre Ziffern interpretiert. Es wird also stets mit Nullen und Einsen gearbeitet.

Operator	Beschreibung	Beispiel	Wert
`&`	Bitweises UND	`A= 9&3`	`A=1`
`^`	Bitweises XODER	`A=9^3`	`A=10`
`|`	Bitweises ODER	`A=9|3`	`A=10`
`<<`	Linksverschiebung	`A=19<<2`	`A=76`
`>>`	Rechtsverschiebung	`A=8>>2`	`A=2`
`>>>`	Rechtsverschiebung (Null-füllend)	`A=18>>>2`	`A=4`

Tabelle 4.7: Bitweise Operatoren

Auch wenn bitweise Operatoren durchaus ihre Berechtigung haben, in normalen JavaScript-Anwendungen werden diese nicht eingesetzt.

4.3.4 Zuweisungsoperatoren

Gleichheitszeichen

Grundsätzlich richtig ist die Aussage, dass lediglich ein Zuweisungsoperator existiert. Bei diesem handelt es sich um das Gleichheitszeichen. Alle anderen sind lediglich Abkürzungen, bei denen das Gleichheitszeichen mit einem bitweisen oder arithmetischen Operator kombiniert wird.

Operator	Beschreibung	Beispiel	Wert
=	Weist dem ersten Operanden den Wert des zweiten Operanden zu.	5 = 4	4
+=	Addiert zwei Zahlen und weist der ersten Zahl das gespeicherte Ergebnis zu.	A+=2	A+2
-=	Subtrahiert zwei Zahlen und weist der ersten Zahl das gespeicherte Ergebnis zu.	A-=5	A-5
*=	Multipliziert zwei Zahlen und weist der ersten Zahl das gespeicherte Ergebnis zu.	A*=4	4*A
/=	Dividiert zwei Zahlen und weist der ersten Zahl das gespeicherte Ergebnis zu.	A=11/=2	A=5.5
%=	Berechnet den Modulo von zwei Operanden und weist dem ersten Operanden das Ergebnis zu.	13 %= 4	1
&=	Es wird eine bitweise UND-Verknüpfung durchgeführt und dem ersten Operanden das Ergebnis zugewiesen.	A&=2	A&2
^=	Es wird eine bitweise XODER-Verknüpfung durchgeführt und dem ersten Operanden das Ergebnis zugewiesen.	3^=5	3^5
|=	Es wird eine bitweise ODER-Verknüpfung durchgeführt und dem ersten Operanden das Ergebnis zugewiesen.	5|=6	5|6
<<=	Bitweise Linksverschiebung	9 <<= 2	36
>>=	Bitweise Rechtsverschiebung (Vorzeichen – Weiterführung)	-13 >>= 2	-4
>>>=	Bitweise Rechtsverschiebung (Null-füllend)	5 >>>= 2	1

Tabelle 4.8: Zuweisungsoperatoren

Auch bei der Verwendung von Zuweisungsoperatoren, ausgenommen **=**, gilt, dass diese nur in Absprache mit Entwickler-Kollegen verwendet werden sollten. Handelt es sich doch hierbei um relativ selten verwendete Schreibweisen, die in dieser Form nicht jedem geläufig sind. Aus stilistischen Erwägungen steht deren Verwendung allerdings nichts im Wege.

4.3.5 Vergleichsoperatoren

wahr oder falsch

Um Werte zu vergleichen, werden Vergleichsoperatoren eingesetzt. Solche Vergleiche sind binäre Operanden, die einen der booleschen Werte **true** und **false** zurückliefern, wobei **true** immer dann geliefert wird, wenn die Aussage wahr, und **false** immer dann geliefert wird, wenn die Aussage unwahr ist. Vergleichsoperatoren werden Ihnen vor allem als Bedingung für Schleifen oder bedingte Abfragen begegnen. Grundsätzlich lassen sich Operatoren nur mit Operanden vom Typ Zahl und vom Typ String anwenden. Als einzige Ausnahme gilt die Überprüfung, ob boolesche Werte oder Objekte gleich oder ungleich sind.

Operator	Beschreibung	Beispiel	Wert
==	Vergleicht zwei Operanden auf ihre Gleichheit.	`9 == 8`	**false**
!=	Vergleicht zwei Operanden auf ihre Verschiedenheit.	`5 == 5`	**false**
===	Überprüft, ob die beiden Operanden gleich groß und vom gleichen Datentyp sind.	`5 === 6`	**false**
!==	Überprüft, ob die beiden Operanden gleich groß oder nicht von gleichem Typ sind.	`5 !== 5`	**true**
>	Prüft, ob der linke Operand größer als der rechte ist.	`34 > 89`	**true**
>=	Prüft, ob der linke Operand größer als der rechte oder gleich groß ist.	`6 >= 6`	**True**
<	Prüft, ob der linke Operand kleiner als der rechte ist.	`5 < 3`	**false**
<=	Prüft, ob der linke Operand kleiner als der rechte oder gleich groß ist.	`8 < 9`	**true**

Tabelle 4.9: Vergleichsoperatoren

Der Vergleich von Zahlen orientiert sich an den reellen Zahlen. So liefert `9 > 2` den Wert **true**, da `9` tatsächlich größer als `2` ist. Auch der Vergleich von Strings ist zulässig. Hier gilt die lexikographische Ordnung. So liefert `abcd < abcde` den Wert **true**, da `abcd` tatsächlich kleiner als `abcde` ist.

4.3.6 String-Operatoren

Durch String-Operatoren können zwei Stings aneinander gehangen werden. Es handelt sich hierbei um die Verknüpfung zweier Zeichenketten.

Bei der Verbindung zweier Strings ist darauf zu achten, dass Leerzeichen zwischen diesen nicht automatisch eingefügt werden. Im Normalfall sollte ein Leerzeichen am Ende des ersten oder zu Beginn des zweiten Strings notiert werden.

Operator	Beschreibung	Beispiel	Wert
+	Verbindet zwei Strings zu einem String.	"Hallo" + "Welt"	"Hallo Welt"
+=	Verbindet zwei Strings zu einem String und schreibt den Wert in die linke String-Variable.	"Hallo"+="Welt"	"Hallo Welt"

Tabelle 4.10: String-Operatoren

4.3.7 Spezielle Operatoren

In JavaScript existiert eine ganze Reihe so genannter „Special Operators". Diese speziellen Operatoren zeigt die folgende Tabelle.

Operator	Beschreibung	Beispiel	Beschreibung
?:	verkürzte Schreibweise für eine **if**-Abfrage	Zeit = "9"; **document.write** ("Aktuelle Zeit: " + (Zeit **?** "9" **:** "12"))	Ist der Wert der Variablen Zeit gleich 9, wird 9 ins Dokument geschrieben. Anderenfalls wird der Wert 12 angezeigt.
,	Trennt mehrere Elemente voneinander.	**for (var** i=0, j=10; i <= 10; i++, j--) **document.writeln** ("a["+i+","+j+"]= " + a[i,j])	Hier werden die beiden Variablen i und j mit einem Mal inkrementiert.
Delete	Löscht ein Objekt, ein Element in einem Array oder eine Objekteigenschaft.	**delete** Autor	Löscht das Objekt Autor.
function	Definiert eine namenlose Funktion.	**var** Autor = **function** (y) {**return** y*y};	Deklariert eine namenlose Funktion. Als Rückgabewert wird das Quadrat von x geliefert.
in	Liefert **true** zurück, wenn die angegebene Eigenschaft in dem spezifizierten Objekt verfügbar ist.	"PI" **in** Math	Liefert **true**, da **PI** eine Eigenschaft des **Math**-Objekts ist.
instanceof	Liefert **true**, wenn das angegebene Objekt ein Objekt des spezifizierten Objekts ist.	Autor = **new** Hornby("Fever Pitch", "1997") neu=Autor **instanceof** Hornby	Liefert **true**, da das Objekt neu ein Objekt von Hornby ist.

Tabelle 4.11: Spezielle Operatoren

Operator	Beschreibung	Beispiel	Beschreibung
New	Erzeugt eine neue Instanz eines vordefinierten oder eines benutzerdefinierten Objekts.	`Autor = new Hornby("Fever Pitch", "1997")`	Erzeugt das neue Objekt `Autor`.
This	Erlaubt den Zugriff auf das aktuelle Objekt.	`<INPUT type="button" value="Los" onclick="alert(this.form.Eingabe.value)">`	Gibt den Wert des Eingabefeldes `Eingabe` aus.
typeof	Liefert einen String, der den Typ des Operanden enthält.	`Typeof('Hallo')`	Liefert den Typ String, da `Hallo` eine Zeichenkette ist.
void	Definierte Rückgabe für Funktionen, die keinen Wert zurückliefern	`<A HREF="javascript:void(0)">Ohne Funktion</A>`	Der Hyperlink ist ohne Funktion

Tabelle 4.11: Spezielle Operatoren (Forts.)

Einige der hier gezeigten Operatoren werden Ihnen im Lauf dieses Buches noch genauer vorgestellt. Andere wiederum spielen in der Praxis kaum eine Rolle. Deren Beschreibung dient daher auch ausschließlich der Vollständigkeit.

4.3.8 Operatoren-Reihenfolge

interne Reihenfolge

Um mit Operatoren korrekt arbeiten zu können, soll hier deren interne Reihenfolge beleuchtet werden. Denn ebenso, wie dies im Umgang mit mathematischen Aufgaben der Fall ist, muss auch in JavaScript-Programmen eine bestimmte Operatoren-Reihenfolge beachtet werden. Auch in JavaScript werden nicht alle Ausdrücke, in denen mehrere Operatoren vorkommen, von links nach rechts gelesen und ausgewertet. Anders jedoch als in der Mathematik, wo die Regel „Punktrechnung geht vor Strichrechnung“ gilt, existieren nicht für alle JavaScript-Probleme solche Regeln. Aus diesem Grund wurde eine Präzedenz, also eine Reihenfolge für JavaScript-Operatoren, an die sich alle interpretierenden Programme halten, festgelegt. Diese Reihenfolge zeigt die folgende Übersicht:

1. Komma (,)
2. Zuweisungen
3. Bedingungen (?:)
4. Logisches ODER
5. Logisches UND
6. Bitorientiertes ODER

7. Bitorientiertes XOR
8. Bitorientiertes UND
9. Gleichheit/Ungleichheit (==, !=)
10. Größer/Kleiner-Beziehungen
11. Bitorientiertes Shift
12. Addition/Subtraktion
13. Multiplikation/Division/Modulo
14. Negation/Inkrementierung/Dekrementierung

Operatoren und Klammern

Nun ist es selbstverständlich so, dass die gezeigte Operatoren-Reihenfolge nicht für jeden Verwendungszweck nützlich ist. Würde es stets bei dieser Reihenfolge bleiben, müssten entweder JavaScript-Programme umgeschrieben werden oder es würden sich bestimmte Probleme gar nicht erst lösen lassen. Damit solcherlei Szenarien vermieden werden können, gibt es die Möglichkeit der Klammerung. Hierzu ein simples Beispiel, in welchem zwei Variablen definiert wurden.

```
var normal = 3 * 5 + 4;
var anders = 3 * (5+4);
```

Die Variable `normal` zeigt eine einfache mathematische Rechnung. Laut der bekannten Regel „Punktrechnung geht vor Strichrechnung" würde JavaScript hier, und dies völlig zu Recht, als Ergebnis `19` ausgeben. Das Ergebnis der Variablen `anders`, welches sich der Klammerung bedient, wäre hier allerdings `27`. Erreicht wird dies durch das Setzen der Klammern, woraus folgt, dass zunächst der Inhalt der Klammer addiert und anschließend das Produkt gebildet wird. Die Klammerung dient jedoch nicht nur der Lösung grundlegender mathematischer Probleme. An Bedeutung gewinnt das Notieren von Klammern vor allem im Zusammenhang mit komplexen Berechnungen und umfangreichen JavaScript-Quellcodes. Durch eine Klammerung lässt sich in vielen Fällen ein weitaus übersichtlicherer Code entwickeln, was wiederum der schnelleren Bearbeitbarkeit des JavaScript-Programms zu Gute kommt.

4.4 Schleifen

Anweisungen mehrmals ausführen

Innerhalb von Programmen kommt es häufig vor, dass Anweisungen mehrmals ausgeführt werden müssen. Um diese Anweisungen nicht stetig zu notieren und somit den Programmier-Aufwand mindern zu können, stehen in JavaScript Schleifen zur Verfügung. Hierbei unterscheidet JavaScript diverse Varianten. Vor der Programmierung eines Programms für eine bestimmte Problemstellung sollte wohl durchdacht werden, ob diese durch den Einsatz von Schleifen überhaupt gelöst werden kann. Der nächste Schritt der Überlegung beschäftigt sich dann mit der Wahl der einzusetzenden Schleife.

4.4.1 for-Schleife

am häufigsten verwendet

Der Vorteil von **for**-Schleifen ist, dass diese Schleifenbedingung in der Regel einen Zähler mit einer gleichzeitigen Abbruchbedingung vorsieht. Eine solche Abbruchbedingung kann beispielsweise sein, dass ein JavaScript-Programm die Quadratzahlen aller numerischen Werte zwischen 1 und 10 ausgibt. Hat das Programm die Zahl 10 erreicht, wird die Funktion beendet. In diesem Fall wäre die Abbruchbedingung demzufolge x > 10. Die **for**-Schleife besitzt die folgende grundlegende Struktur, in der drei Parameter fest verankert sind:

```
for (Initial-Ausruck; Bedingung; Änderung)
{ Anweisungen }
```

Der hier als `Initial-Ausdruck` gekennzeichnete Parameter deklariert eine Zählvariable, welche die Anzahl der von dem Programm zu durchlaufenden Wiederholungen zählt. Für den Fall, dass mehrere dieser Variablen vorkommen, müssen diese jeweils durch ein Komma voneinander getrennt werden. Der in der allgemeinen Syntaxform als `Bedingung` gekennzeichnete Paragraph gibt die für die Schleife festzulegende Abbruchbedingung an. Diese könnte also, um bei dem eingangs erwähnten Quadrat-Beispiel zu bleiben, `x > 10` sein. Nach jedem erfolgten Durchlauf der Schleife wird die Anweisung `Änderung` ausgeführt. Hier muss die Laufvariable so verändert werden, dass die Schleifenbedingung auch tatsächlich erfüllt werden kann.

Aufwand verringern

Wollte man die Quadrate aller Zahlen zwischen 0 und 20 in einem Dokument ausgeben, wäre dies, durch HTML-Syntax realisiert, ein enormer Aufwand. Wie groß müsste dann erst dieser Aufwand sein, wenn beispielsweise das Quadrat aller Zahlen zwischen 1 und 1.000 ermittelt werden sollte? Anstelle dieses denkbar schlechten HTML-Weges soll Ihnen das folgende Beispiel zeigen, wie vorzüglich sich **for**-Schleifen für diesen Zweck eignen. Die aufgeführte Syntax hat zur Folge, dass die Quadrate der Zahlen zwischen 0 und 20 dynamisch in das Dokument geschrieben werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function zeigeQuadrat()
{
for (var i=0; i<=20; i++)
{
document.write("Das Quadrat von " + i + " ist: ")
document.write(i*i + "<BR>")
}
}
```

```
</SCRIPT>
</HEAD>
<BODY onload="zeigeQuadrat()">
</BODY>
</HTML>
```

Listing 4.6: Ausgabe der Quadratzahlen von 0 bis 20

Die Funktion `zeigeQuadrat()` wird durch den Event-Handler **`onload()`** aufgerufen. Innerhalb der Funktion befindet sich eine **`for`**-Schleife. Zunächst wird die Laufvariable `i` deklariert und mit dem Wert `0` initialisiert. Im nächsten Schritt wird überprüft, ob die Bedingung `i` **`<=`**`20` erfüllt ist. Ist dies der Fall, führt JavaScript die Anweisung `i*i` aus. Der erste Schleifendurchlauf ist somit abgeschlossen. Im nächsten Schritt wird die Laufvariable `i` durch die Anweisung `i`**`++`** um den Wert eins erhöht. Auch hieran schließt sich die Überprüfung der Schleifendbedingung, ob `i`**`<=`**`20 ist`, an. Dieser Vorgang wird so lange wiederholt, bis die Schleifenbedingung nicht mehr erfüllt wird.

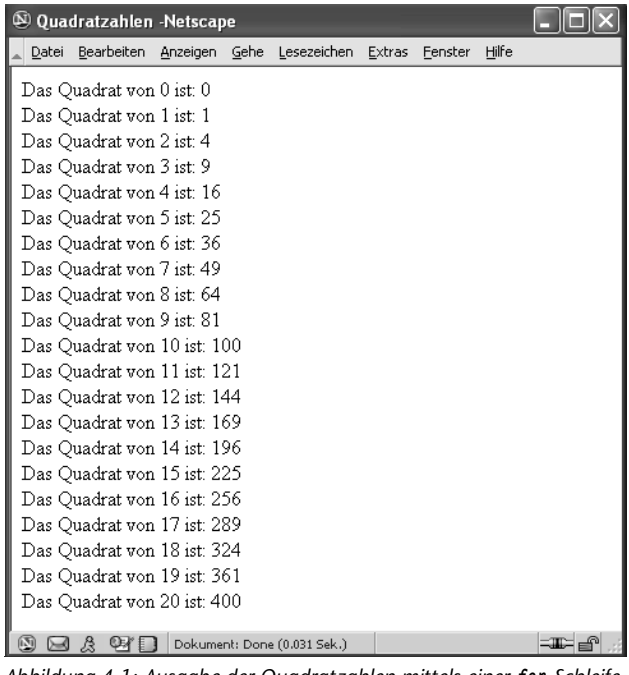


Abbildung 4.1: Ausgabe der Quadratzahlen mittels einer `for`-Schleife

4.4.2 while-Schleife

vereinfachte Syntax

Die Syntax einer **while**-Schleife ist simpler als die einer **for**-Schleife. Mittels der **while**-Schleife kann ein Befehl bzw. ein Befehlsblock so lange ausgeführt werden, bis die Bedingung der Schleife erfüllt ist. Die **while**-Schleife eignet sich demnach für solche Anwendungen, bei denen die Anzahl der Schleifendurchläufe im Vorfeld nicht bekannt ist. Beachten Sie, dass die **while**-Schleife in jedem Fall entweder ein erfüllbares Abbruchkriterium oder aber die **break**-Anweisung beinhalten sollte. Nur so lassen sich Endlosschleifen, die häufig zu Systemabstürzen führen, vermeiden. Die folgende allgemein gültige Syntax beschreibt eine **while**-Schleife:

```
while ( bedingung )
{ Anweisung }
```

Hinter dem Schlüsselwort **while** wird die Bedingung notiert. Ist diese erfüllt, wird die **while**-Schleife mindestens einmal ausgeführt. Hieran schließt die auszuführende Anweisung an. Die nachstehende Beispiel-Syntax beschreibt, wie sich die Quadrate der Ziffern 1 bis 25 mittels einer **while**-Schleife ermitteln und anhand der **write()**-Methode dynamisch in das Dokument schreiben lassen.

```
<SCRIPT type="text/JavaScript">
<!--
var x,i;
i=1;
while(i<=25)
{
x=i*i;
document.write("<BR>Das Quadrat von "+ i +" ist " + x);
 ++i;
}
//-->
</SCRIPT>
```

Zunächst werden die beiden Variablen x und i deklariert. Die Variable i wird anschließend mit dem Wert 1 initialisiert. Hinter dem Schlüsselwort **while** wird die Schleifenbedingung notiert. Diese überprüft, ob i**<=**25 ist. Innerhalb der Schleife wird der Variablen x der Wert i*****i zugewiesen. Der hier ermittelte Wert wird in das Dokument geschrieben. Abschließend wird der Wert der Variablen i um jeweils eins erhöht. Die Schleife wird so oft durchlaufen, bis die Bedingung i**<=**25 nicht mehr erfüllt ist.

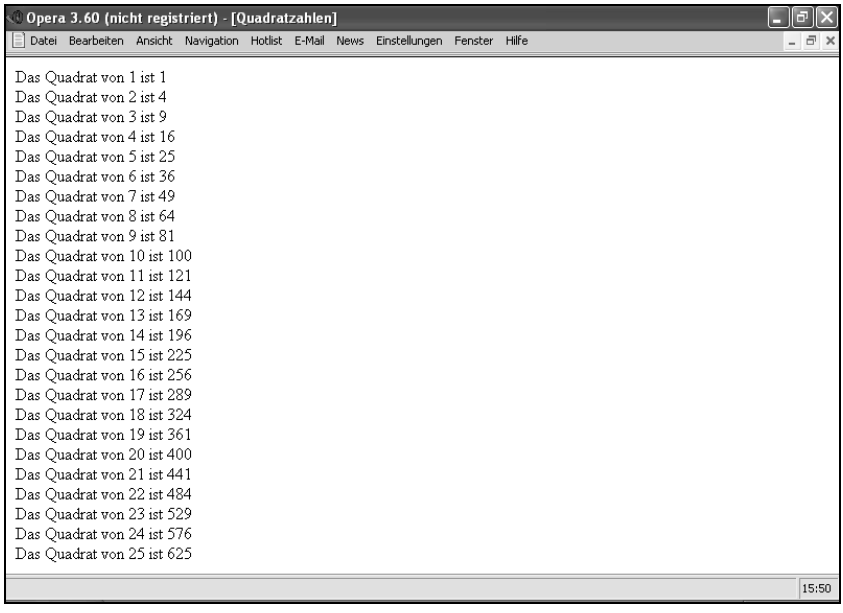


Abbildung 4.2: Die Ausgabe der Quadratzahlen mittels einer `while`-Schleife

4.4.3 do-while-Schleife

Die Schleife wird mindestens einmal ausgeführt.

Prinzipiell handelt es sich bei der **`do-while`**- um die gleiche wie die **`while`**-Schleife. Ein gravierender Unterschied besteht aber dennoch: Während bei der **`while`**-Schleife vor dem ersten Durchlauf die Schleifendbedingung überprüft wird, geschieht dies bei der **`do-while`**-Schleife erst danach. Hieraus folgt, dass die **`do-while`**-Schleife in jedem Fall einmal ausgeführt wird. Die folgende allgemein gültige Syntax kennzeichnet eine **`do-while`**-Schleife:

```
do {Anweisung}
while (Bedingung);
```

Hinter dem Schlüsselwort **`do`** folgt die auszuführende Anweisung. Erst hieran schließt sich die Schleifenbedingung an. Diese wird hinter dem Schlüsselwort **`while`** notiert. Die nachstehende Syntax zeigt, wie sich die Quadrate der Zahlen 1 bis 25 mittels einer **`do-while`**-Schleife ermitteln und dynamisch in das Dokument schreiben lassen.

```
<SCRIPT type="text/JavaScript">
<!--
var x,i;
i=1;
do
{
```

```
 x=i*i;
document.write("<BR>Das Quadrat von "+ i +" ist " + x);
 ++i;
}
while(i<=25)
//-->
</SCRIPT>
```

Innerhalb des Script-Bereichs werden die beiden Variablen `x` und `i` definiert. Die Variable `i` wird mit dem Wert `1` initialisiert. Hinter dem Schlüsselwort **`do`** folgt der Anweisungsblock. In diesem wird die Variable `x` mit dem Wert `i*i` initialisiert. Der ermittelte Wert wird über die **`write()`**-Methode ausgegeben und der Zähler anschließend durch die Anweisung **`++`**`1` um den Wert eins erhöht. Die Schleifenbedingung wird hinter dem Schlüsselwort **`while`** in Klammern gesetzt notiert. In diesem Beispiel besagt die Bedingung, dass die Schleife so oft durchlaufen werden soll, wie `i<=25` ist.

4.4.4 for..in-Schleife

Eigenschaften von Objekten ermitteln

Über die **`for..in`**-Schleife können alle Eigenschaften eines Objekts ermittelt werden. So lassen sich beispielsweise alle Eigenschaften des **`navigator`**-Objekts auslesen. Ein Abbruchkriterium kann bei dieser Schleife nicht angegeben werden. Die **`for..in`**-Schleife wird so oft durchlaufen, bis alle Eigenschaften eines Objekts ermittelt sind. Die folgende allgemein gültige Syntax beschreibt den Aufbau einer **`for..in`**-Schleife:

```
For ( Eigenschaft in Objekt )
Anweisung
```

Hinter dem einleitenden Schlüsselwort **`for`** wird in Klammern gesetzt eine Variable notiert. Hieran schließt sich die Notation des **`in`**-Schlüsselworts, gefolgt von dem entsprechenden Objekt, an. Es folgt die Notation der ausführenden Anweisung. Die **`for..in`**-Schleife ist vor allem dann sinnvoll, wenn nicht bekannt ist, wie viele und welche Eigenschaften ein Objekt besitzt. Im folgenden Beispiel werden alle Eigenschaften des **`navigator`**-Objekts mittels der **`write()`**-Methode dynamisch in das Dokument geschrieben.

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

Innerhalb des Script-Bereichs wird zunächst die Variable `wert` deklariert. Die Schleife wird durch das Schlüsselwort **`for`** eingeleitet. Innerhalb der Klammer wird die Variable `Eigenschaft` deklariert. Hieran schließen sich die Notation des **`in`**-Schlüsselworts sowie des **`navigator`**-Objekts an. Der Klammerausdruck lässt sich folgendermaßen übersetzen: Ermittle alle `Eigenschaften` in dem Objekt **`navigator`**. Der Anweisungsblock dient dazu, die ermittelten Eigenschaften dynamisch in das Dokument zu schreiben, wobei zunächst die Eigenschaft und anschließend deren Werte notiert und über die **`write()`**-Methode ausgegeben werden.

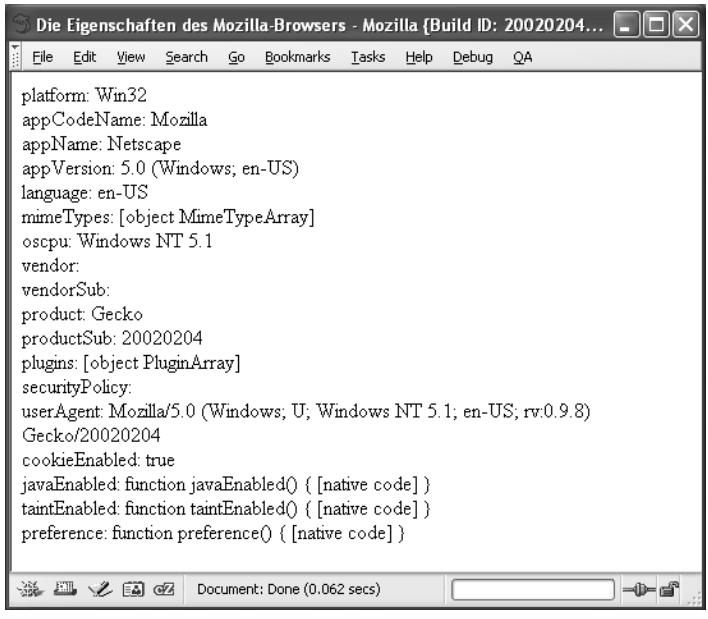


Abbildung 4.3: Alle Eigenschaften des `navigator`-Objekts im Mozilla-Browser

4.4.5 break und continue

Schleifen steuern

Es ist nun nicht in jedem Fall so, dass eine Schleife exakt so oft durchlaufen wird, wie es die eigentliche Funktion erwarten lassen würde. Zum einen kann dies an der Komplexität des JavaScript-Programms liegen. Häufig spielen hierbei jedoch auch andere Faktoren eine entscheidende Rolle und sei es eben nur der unbedingte Wille des Programmierers, auf Grund einer bestimmten Bedingung direkt zum nächsten Schleifendurchlauf zu springen. JavaScript kennt zwei zusätzliche Befehle, um Schleifen innerhalb eines Programms gezielt steuern zu können.

- **`break`** – bricht die Schleifenbedingung ab
- **`continue`** – springt zum nächsten Schleifendurchlauf

seltener Praxiseinsatz

Obwohl sicherlich auch diese beiden gesonderten Formen der Schleifenbehandlung in Einzelfällen eine Daseinsberechtigung besitzen, werden diese in der Praxis nur äußerst selten eingesetzt. Dies mag zum einen daran liegen, dass sich mögliche Einsatzgebiete nur in einigen wenigen Ausnahmen finden lassen, schwerer wiegt jedoch, dass es sich bei diesen Lösungen nicht um die elegantesten Varianten handelt. Aber dennoch: Trotz dieser Einschränkungen soll der Einsatz von **break** und **continue** hier vorgestellt und erläutert werden. Die erste Syntaxform beschreibt den Einsatz der **break**-Anweisung. Deren Einsatz hat zur Folge, dass eine integrierte Schleife augenblicklich beendet wird. In dem gezeigten Beispiel wird die Abhängigkeit zwischen einer definierten Bedingung und der **break**-Anweisung deutlich. Die hier verwendete **while**-Schleife ließe auf Grund der Notation das Hochzählen bis zum Wert `20` zu. Durch den Einsatz des **break**-Schlüsselwortes wird diese eigentlich ausdrücklich gestattete Vorgehensweise jedoch unterbunden und stattdessen das Hochzählen bis zum Wert `6` zugelassen. Ist dieser Wert erreicht, folgt die Ausgabe in einem **alert()**-Fenster.

```
<SCRIPT type="text/JavaScript">
var i = 0;
while (i < 20)
{
if ( i ==6) break;
i++;
}
alert("i = " + i);
</SCRIPT>
```

Die Funktionalität und Syntax der **break**-Anweisung wird auf den ersten Blick deutlich. Innerhalb der Schleife, in diesem Beispiel handelt es sich um eine **while**-Schleife, wird die **if**-Abfrage notiert. Innerhalb derer wiederum muss das Schlüsselwort **break** angegeben werden. Allein auf Grund dieser Notation ergibt sich in diesem Beispiel der Abbruch der **while**-Schleife, wenn der Wert `6` erreicht ist, obwohl der eigentlichen Schleife die Bedingung `20` zugewiesen wurde. Im Zusammenhang mit der **break**-Anweisung existiert eine gesonderte Notationsform, deren Einsatz sich allerdings auf komplexe Funktionen mit verschachtelten **if**-Abfragen beziehen sollte. Zunächst folgt ein einfaches Beispiel, anhand dessen die allgemeine Syntaxform abgelesen werden kann.

```
<SCRIPT type="text/JavaScript">
var i = 0;
while (i < 20)
{
Abbruch:
if ( i ==6)
```

```
{
alert("i = " + i);
break Abbruch;
}
i++;
}
</SCRIPT>
```

Das Beispiel erledigt die gleichen Aufgaben, die bereits innerhalb der vorangegangenen Syntax exemplarisch dargestellt wurden. Der Unterschied liegt hier jedoch im Detail verborgen und reduziert sich auf die abgewandelte Syntaxform. So ist es freilich auch wenig verwunderlich, dass dieses Beispiel den Ansprüchen einer geeigneten Anwendung nicht gerecht wird. Aber wie dem auch sei: An dieser Stelle geht es schlicht und ergreifend um die Art der Notation. Denn hier wird, anders als bei der schon vorgestellten Variante, vor der **if**-Abfrage ein so genanntes Label notiert. Hierbei handelt es sich um einen frei wählbaren Namen, gefolgt von einem abschließenden Doppelpunkt. Bei der Wahl des Namens gelten die bekannten Regeln für die Wahl von Bezeichnern in JavaScript. In diesem Beispiel wurde das Label `Abbruch` spezifiziert. Auf dieses Label Bezug nehmend, wird innerhalb der **if**-Abfrage das Schlüsselwort **break** mit dem dazugehörigen Label, hier ist dies also `Abbruch`, angegeben. Durch diese Vorgehensweise ist sichergestellt, dass sich die **break**-Anweisung exakt auf die gewünschte **if**-Abfrage bezieht. Hierbei spielt es keine Rolle, wie viele verschachtelte **if**-Abfragen vorkommen.

den nächsten Schleifendurchlauf ansteuern

JavaScript bietet durch die **continue**-Anweisung die Möglichkeit, direkt zum nächsten Schleifendurchlauf zu springen. Diese Vorgehensweise kann immer dann sinnvoll sein, wenn von vornherein feststeht, dass der zu überspringende Durchlauf nicht das für das Programm relevante Ergebnis liefert. Wurde innerhalb einer Schleife die **continue**-Anweisung notiert, werden die nachfolgenden Anweisungen, die sich innerhalb dieser Schleife befinden, während des aktuellen Schleifendurchlaufs nicht mehr ausgeführt. Auch für diesen Fall soll ein einfaches Beispiel einen ersten Einblick ermöglichen. Im folgenden Beispiel sind die zwei Variablen `a` und `b` als Zähler definiert. Deren Zählweise wird bei jedem Schleifendurchlauf um `1` erhöht. Dies wird für den Zeitraum fortgesetzt, bis der Zähler `a` einen Wert von `20` besitzt. Die integrierte **if**-Abfrage stellt fest, ob der Wert `a` während der Durchläufe den Wert `5` erhält. Ist dies der Fall, wird automatisch der nächste Schleifendurchlauf gestartet. Demzufolge kann die Anweisung `b`**++** während dieses Schleifendurchlaufs nicht mehr ausgeführt werden. Dies führt dazu, dass nach dem kompletten Durchlaufen der Schleife in einem **alert()**-Fenster für `a` ein Wert von `20`, für `b` jedoch nur ein Wert von `19` angezeigt wird.

```
<SCRIPT type="text/JavaScript">
var a = 0;
var b = 0;
while (a < 20)
{
a++;
if (a ==5) continue;
b++;
}
alert ("a hat den Wert " + a + " und b hat den Wert " + b );
</SCRIPT>
```

Im Zusammenhang mit der **continue**-Anweisung bleibt mir im Endeffekt nur eine ähnliche Aussage, wie ich sie bereits während der **break**-Ausführung getätigt habe. Für komplexe Anwendungen kann **continue** sicherlich eine gute Alternative gegenüber herkömmlichen Notationsformen darstellen. Bei simplen Funktionen, und eben eine solche war das vorherige Beispiel, kann und sollte hierauf jedoch verzichtet werden.

4.5 Fallunterscheidung

JavaScript wäre längst nicht so leistungsfähig, wenn in dieser Sprache nicht die Fallunterscheidung möglich wäre. Es stehen verschiedene Varianten dieser Unterscheidung zur Verfügung. Der Vorteil von Fallunterscheidungen gegenüber anderen Syntaxformen besteht zweifelsohne in der weitaus simpleren Handhabbarkeit von Fallunterscheidungen. Zudem bietet JavaScript neben einfachen Entweder-Oder-Abfragen auch Fallunterscheidungen, mit deren Hilfe exakt zwischen mehreren Fällen unterschieden werden kann.

4.5.1 if-Abfrage

Wenn-Dann-Bedingung

Die **if**-Abfrage bietet die Möglichkeit, Aktionen in Abhängigkeit von der Erfüllung einer Bedingung auszulösen. Diese bedingte Anweisung wird mit dem Schlüsselwort **if** eingeleitet. Hieran schließen sich die Bedingung in Form eines logischen Ausdrucks sowie die auszuführende Aktion an. Um bei Nichterfüllung der Bedingung einen anderen Zweig des Programms anzuwenden folgt das Schlüsselwort **else**, ebenfalls gefolgt von der auszuführenden Aktion. Folgende allgemein gültige Syntax wird für die **if**-Abfrage verwendet.

```
if (Bedingung)
{
dann-Anweisungen
}
```

```
else
{
sonst-Anweisungen
}
```

Die **if**-Anweisung kann beinahe wörtlich übersetzt werden: Ist die Bedingung erfüllt, dann führe die `dann-Anweisung` aus, ist sie nicht erfüllt, führe die `sonst-Anweisung` aus. Beachten Sie, dass bei der **if**-Abfrage nicht zwingend ein **else**-Zweig vorhanden sein muss. In ihrer einfachsten Syntax stellt sie sich vielmehr folgendermaßen dar:

```
if (Bedingung)
{
dann-Anweisungen
}
```

optionaler else-Zweig

Wenn auch optional, sollte der **else**-Zweig in den meisten Programmen vorhanden sein. Hierin sollte eine Anweisung notiert werden, die auf den Fall reagiert, dass die **if**-Bedingung nicht erfüllt ist. Das folgende JavaScript zeigt Ihnen die Realisierung eines Passwortschutzes. Zu diesem Zweck wird hier die **if**-Abfrage einschließlich eines **else**-Zweigs verwendet. Das Passwort lautet `Kugel`. Wird dieses korrekt eingegeben, wird der Text `Das ist korrekt` in das Dokument geschrieben. Anderenfalls wird der Text `Das ist nicht korrekt` angezeigt.

```
<SCRIPT type="text/JavaScript">
<!--
var Passwort = "Kugel"
var Eingabe = window.prompt
("Geben Sie bitte Ihr Passwort ein", "");
if(Eingabe == Passwort)
{
document.write("Das ist korrekt")
}
else
{
document.write("Das ist nicht korrekt");
}
//-->
</SCRIPT>
```

Die Bedingung der **if**-Abfrage lautet: Ist der Wert `inhalt` gleich dem Wert `pass`? Um es nochmals anhand der fast wörtlichen Übersetzung zu erklären: Ist diese Bedingung erfüllt, führe die Anweisung **document.write(**"`Das ist korrekt`"**)** aus. Ist diese Bedingung nicht erfüllt, dann führe die Anweisung **document.write(**"`Das ist nicht korrekt`"**)** aus.

Verschachtelte if-Anweisungen

mehrere **if**-*Abfragen*

Es ist möglich, mehrere **if**-Abfragen zu verschachteln. Hierdurch kann in einem Ergebniszweig eine zweite **if**-Abfrage notiert werden. Dies kann hinter einem anderen **if**- bzw. **else**-Zweig angegeben werden. Somit lassen sich mehrere Bedingungen gleichzeitig überprüfen. Im folgenden Beispiel wird kontrolliert, ob drei Formularfelder den gleichen Wert besitzen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Wert()
{
var inhalt = document.formular.eingabe.value;
var inhalt2 = document.formular.eingabe2.value;
var inhalt3 = document.formular.eingabe3.value;
if (inhalt == inhalt2)
{
  if (inhalt == inhalt3)
   {
      if (inhalt2 == inhalt3)
        {
          alert("Eingabe korrekt");
        }
   }
}
else alert("Falsche Eingabe");
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="formular">
<INPUT type="text" name="eingabe" value="">
<INPUT type="text" name="eingabe2" value="">
<INPUT type="text" name="eingabe3" value="">
</FORM>
<A href="javascript:Wert()">pr&uuml;fe</A>
</BODY>
</HTML>
```

Listing 4.7: Verschachtelte `if`*-Abfragen*

Innerhalb des Beispiels werden die Werte der drei Eingabefelder mittels dreier **if**-Abfragen überprüft. Nur wenn alle drei Werte tatsächlich identisch sind, wird der Text `Eingabe korrekt` über die **alert()**-Methode ausge-

geben. Sollte ein oder sollten zwei bzw. drei Werte voneinander abweichen, wird der Text `Falsche Eingabe` ausgegeben. Beachten Sie, dass bei verschachtelten **if**-Anweisungen die Klammersetzung schnell unübersichtlich wird. Dem sollten Einrückungen entgegengesetzt werden.

Alternative Abfrage

verkürzte Schreibweise

Sie können **if**-Abfragen vereinfachen und somit den Quelltext straffen. Dies ist vor allem immer dann ratsam, wenn es sich um einfache **if**-Abfragen handelt, bei denen nur eine Entweder-Oder-Entscheidung vonnöten ist. Nachstehend sehen Sie die allgemein gültige Syntax für eine alternative **if**-Abfrage.

```
(Bedingung) ? Wert1 : Wert2
```

Jede alternative **if**-Abfrage wird durch die Notation einer Bedingung in Klammern eingeleitet. Hieran schließt sich das Fragezeichen an. Anschließend wird derjenige Wert notiert, welcher dann aktuell ist, wenn die Bedingung erfüllt ist. Hinter dem Doppelpunkt wird der Wert angegeben, welcher dann aktuell ist, wenn die Bedingung nicht erfüllt ist. Schauen wir uns nochmals das Beispiel der Passwort-Abfrage vom Beginn dieses Kapitels an. Dieses Mal wird hier jedoch die alternative Schreibweise angewandt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Passwort = "Kugel"
Eingabe = window.prompt
("Geben Sie bitte Ihr Passwort ein", "");
(Eingabe == Passwort) ? document.write("Das ist korrekt") : docu-
ment.write("Das ist nicht korrekt");
//-->
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.8: Die Passwort-Abfrage in alternativer Schreibweise

Die Variablendeklaration `Passwort` beinhaltet das gesuchte Passwort `Kugel`. Um ein Eingabefenster zu generieren, werden der Variablen `Eingabe` das **window**-Objekt sowie dessen **prompt()**-Methode und der im Fenster anzuzeigende Text zugewiesen. Die Bedingung `Eingabe == pass`, die überprüft, ob der Wert des Fensterinhalts mit dem Passwort überein-

stimmt, wird in Klammern notiert. Ist diese Bedingung erfüllt, wird der Text `Das ist korrekt` dynamisch in das Dokument geschrieben. Für den Fall, dass die Bedingung nicht erfüllt ist, wird der Text `Das ist nicht korrekt` angezeigt.

4.5.2 Switch-Abfrage

mehrere Fälle unterscheiden

Häufig muss zwischen mehreren Fällen unterschieden werden. Zwar haben Sie bereits die Möglichkeit zum Verschachteln von **`if`**-Abfragen kennen gelernt, diese Variante ist jedoch nicht sonderlich komfortabel. Für diesen Zweck besser geeignet ist die **`switch`**-Abfrage. Beachten Sie, dass diese Abfrage erst seit JavaScript 1.2 zur Verfügung steht. Ältere Browser können eine solche Syntax nicht interpretieren. Folgendes allgemein gültige Syntaxkonstrukt beschreibt die Form einer **`switch`**-Abfrage.

```
switch (Ausdruck)
{
case Wert1:
Anweisungen;
break
case Wert2:
Anweisungen;
break
case Wert3:
Anweisungen;
break
}
```

Abfragen werden durch **`break`** *beendet.*

Jede **`switch`**-Abfrage wird durch das Schlüsselwort **`switch`** eingeleitet. Innerhalb der Klammern wird der zu überprüfende Wert notiert. Jeder Fall wird durch das Schlüsselwort **`case`** eingeleitet. Hieran schließt sich die Notation des entsprechenden Wertes an. Im nächsten Schritt wird die auszuführende Anweisung notiert. Abgeschlossen wird ein **`case`**-Block durch das Schlüsselwort **`break`**. Hierdurch wird dem Interpreter mitgeteilt, dass die folgenden Anweisungen nicht abgearbeitet werden müssen und der aktuelle **`switch`**-Block verlassen werden kann. Da nicht zwingend einer der definierten Fälle auftritt, sollte im letzten Schritt festgelegt werden, wie in einem solchen Fall vorgegangen werden soll. Zu diesem Zweck steht das Schlüsselwort **`default`** zur Verfügung. Die hier zugewiesene Anweisung wird nun immer dann ausgeführt, wenn keiner der Fälle erfüllt ist. Das nachstehende Script soll die Definition einer **`switch`**-Anweisung demonstrieren. Dem Anwender wird ein Eingabefenster mit der Aufforderung, einen Wert zwischen `1` und `4` einzugeben, angezeigt. Wurde ein gültiger Wert notiert, wird dieser mittels der **`alert()`**-Methode ausgegeben. Für den Fall, dass ein anderes Zeichen

oder eine Ziffer, die nicht zwischen 1 und 4 liegt, eingegeben wird, erscheint in einem Meldungsfenster die Nachricht keine gültige Ziffer.

```
<SCRIPT type="text/JavaScript">
<!--
Ziffer = window.prompt("Ziffer zwischen 1 und 4","");
switch(Ziffer)
{
case "1":
alert("Ziffer1");
break;
case "2":
alert("Ziffer2");
break;
case "3":
alert("Ziffer3");
break;
case "4":
alert("Ziffer4");
break;
default:
alert("keine gültige Ziffer");
break;
}
//-->
</SCRIPT>
```

Es wird für jeden Fall überprüft, ob der Wert mit einem der aufgeführten Fälle identisch ist. Die Vorteile dieser Syntax liegen auf der Hand. Zum einen müssen keine komplexen Verschachtelungen wie bei der **if**-Abfrage vorgenommen werden und zum anderen ist diese Syntax sehr viel leichter zu handhaben. Zusätzlich handelt es sich bei der Verwendung der **switch**-Abfrage um guten Programmierstil. Schließlich spricht man von guten Programmen, wenn für ein bestimmtes Problem die beste und effizienteste Lösung gefunden wurde.

4.6 Funktionen

Wie in vielen anderen Programmiersprachen werden auch in JavaScript Funktionen als fundamentaler und hervorstechender Bestandteil verstanden. Funktionen sind auf Grund ihrer Syntax leicht verständlich und dienen einer wohl durchdachten und ausgereiften Strukturierung von JavaScript-Programmen. Ein Programmblock, der innerhalb einer Funktion spezifiziert ist, wird nicht sofort ausgeführt, kann aber stets aufgerufen werden. Jede Funktion ist ein Anweisungsblock und kann an einer beliebigen Stelle innerhalb des JavaScript-Bereichs oder in einer

externen JavaScript-Datei notiert werden. Eine Funktion muss einen dateiweit eindeutigen Namen besitzen, der bestimmten Konventionen unterliegt.

Konventionen für Funktionsnamen

- Funktionsnamen beginnen mit einem Buchstaben oder einem Unterstrich. Deutsche Umlaute werden zwar von den großen Browsern akzeptiert, sollten aber dennoch nicht verwendet werden.
- Funktionsnamen können aus Buchstaben, dem Unterstrich und ganzzahligen Ziffern bestehen.
- Zahlen sind zulässig, jedoch nicht als erstes Zeichen.

Jede Funktion besitzt einen eindeutigen Namen und enthält einen Anweisungsblock. So kann im denkbar einfachsten Fall eine häufig verwendete Anweisung als Funktion definiert und an den verschiedensten Stellen des JavaScript-Programms aufgerufen werden. Einen solchen Fall beschreibt die nachstehend notierte Funktion `Welt()`. Diese gibt in einem Meldungsfenster die Zeichenkette `Hallo Welt` aus.

```
function Welt()
{
alert("Hallo Welt")
}
```

Der Programmierer kann die Funktion an jeder Stelle des Dokuments durch die Notation des Funktionsnamens `Welt()` aufrufen. Diese Funktionsart erschließt jedoch bei weitem nicht alle Facetten von JavaScript-Funktionen. Interessant sind im Umgang mit Funktionen vor allem die Möglichkeiten der Zuweisung von Parametern sowie des Zurücklieferns von Funktionswerten. Mehr zu diesen Themen erfahren Sie im Laufe dieses Kapitels. Zunächst der grundlegende Aufbau einer JavaScript-Funktion.

```
function name ([parameter_1] [, parameter_2] […,parameter_n])
{
// Programm-Anweisungen
return
}
```

Jede Definition einer JavaScript-Funktion wird durch das Schlüsselwort **`function`** eingeleitet. Hieran schließt sich, durch ein Leerzeichen getrennt, der Funktionsname an. Für diesen gelten die genannten Konventionen. Im einfachsten Fall folgt hinter dem Funktionsnamen eine öffnende und schließende Klammer. Besitzt die Funktion Parameter, werden diese innerhalb der Klammer notiert, wobei mehrere Parameter jeweils durch ein Komma voneinander getrennt werden müssen.

4.6.1 Funktionen definieren

eine erste Funktion

Nachdem wir die grundlegende Syntax von JavaScript-Funktionen bereits betrachtet haben, soll die Definition von Funktionen anhand eines Beispiels verdeutlicht werden. In diesem Beispiel wird dem Anwender ein Eingabefenster angezeigt. In diesem wird als Wert ein URL erwartet. Nach dem Bestätigen des Eingabewertes wird der eingegebene URL im Browserfenster aufgerufen. Zunächst die Syntax:

```
<SCRIPT type="text/JavaScript">
<!--
function ZielURL()
{
var Url = window.prompt("Geben Sie einen URL an:","");
window.location.href = Url;
}
//-->
</SCRIPT>
```

Hinter dem einleitenden Schlüsselwort **function** wird der Funktionsname `ZielURL()` notiert. Innerhalb der geschweiften Klammern folgt der Anweisungsblock. Dieser beinhaltet in diesem Beispiel die Variable `Url`. Als Wert wird dieser die **prompt()**-Methode des **window**-Objekts zugewiesen. Dies bewirkt, dass ein Eingabefenster geöffnet wird. Im nächsten Schritt wird der **href**-Eigenschaft des **location**-Objekts der Wert der Variablen `Url` zugewiesen. Beachten Sie, dass die geschweifte Klammer nach Beendigung des Anweisungsblocks in jedem Fall wieder geschlossen werden muss.

4.6.2 Funktionen aufrufen

identische Schreibweisen

Nach der Definition einer Funktion sollte diese, um das JavaScript-Programm auszulösen, aufgerufen werden. Um einen Funktionsaufruf zu realisieren, muss der Funktionsname in exakt der gleichen Schreibweise verwendet werden. Funktionen können auf unterschiedliche Weise aufgerufen werden.

```
<A href="javascript:Funktionsname()">Verweistext</A>
<A href="#" onclick="javascript:Funktionsname()"> Verweistext</A>
```

Die erste Variante zeigt, dass dem **href**-Attribut das Schlüsselwort **javascript** sowie durch Doppelpunkt hiervon getrennt der Funktionsname, gefolgt von einer öffnenden und schließenden Klammer, zugewiesen wird. Die gleiche Syntax kann auch auf eine Notation angewandt werden, die sich eines Event-Handlers bedient. Um den Quellcode zu optimieren, kann diese Syntax, dies gilt jedoch nur im Zusammenhang mit

Event-Handlern, verkürzt werden. Eine vereinfachte Syntax stellt sich beispielsweise folgendermaßen dar:

```
<INPUT type="button" value="prüfen" onclick="Funktionsname()">
```

Im Zusammenhang mit Event-Handlern (und nur hier!) kann auf das Schlüsselwort **javascript** vor dem Funktionsnamen verzichtet werden. Die folgende Syntax beschreibt exemplarisch, wie die Funktion `Wechsler()` aufgerufen wird. Diese Funktion, obwohl an dieser Stelle nicht weiter auf die Syntax eingegangen werden soll, dient dazu, den Hintergrund der Seite variabel zu gestalten.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Wechsler()
{
 if(document.the_form.farbe[0].checked == true)
  document.all.hinten.style.backgroundColor = "yellow";
 else if(document.the_form.farbe[1].checked == true)
  document.all.hinten.style.backgroundColor = "green";
 else
  document.all.hinten.style.backgroundColor = "blue";
}
//-->
</SCRIPT>
</HEAD>
<BODY id="hinten">
<FORM name="the_form">
<INPUT type="radio" name="farbe" value="gelb">Gelb
<INPUT type="radio" name="farbe" value="gr&uuml;n">Gr&uuml;n
<BR>
<A href="javascript:Wechsler()">W&auml;hlen Sie eine Hintergrundfarbe</A>
</FORM>
</BODY>
</HTML>
```

Listing 4.9: Aufruf der Funktion `Wechsler()`

Der Funktionsaufruf wird hier über einen Hyperlink und ohne die Verwendung eines Event-Handlers realisiert. Demzufolge muss in jedem Fall das Schlüsselwort **javascript** verwendet werden. Hieran schließen sich der Doppelpunkt, der Funktionsname sowie die öffnende und schließende Klammer an. Beachten Sie, dass die Schreibweise des Funktionsnamens in jedem Fall gleich sein muss. Zwar unterscheidet HTML nicht zwischen Groß- und Kleinschreibung, JavaScript tut dies aber sehr

wohl. So würde beispielsweise die Syntax **javascript**:wechsler**()** nicht zu dem gewünschten Ergebnis führen. Wurde doch hier der Funktionsname mit einem Kleinbuchstaben eingeleitet, was der Definition innerhalb des Script-Bereichs nicht entspricht.

4.6.3 Funktionen mit Parametern definieren

call by value und call by reference

Funktionen können ein oder mehrere Parameter übergeben werden. Hierbei wird zwischen zwei verschiedenen Varianten, nämlich call by value und call by reference, unterschieden. Wird der Funktion der Wert einer Variablen als Parameter übergeben, so spricht man hier von call by value. Die Funktion legt hierbei eine Kopie der Variablen an. Wird nun innerhalb des Anweisungsblocks der Funktion der Parameter mit einem neuen Wert belegt, wirkt sich dies nicht auf die Werte der Variablen innerhalb des Hauptprogramms bzw. innerhalb einer anderen Funktion aus. Bei call by reference wird ein Objekt und keine Variable als Parameter an die Funktion übergeben. Auf Grund der Komplexität von Objekten empfiehlt es sich, nicht eine Kopie des Objekts, sondern eine Referenz, die auf das Objekt zeigt, als Parameter der Funktion zu übergeben. Da in diesem Fall nicht mit einer Kopie, sondern mit dem eigentlichen Objekt gearbeitet wird, wirken sich sämtliche Änderungen innerhalb der Funktion auch auf das Hauptprogramm bzw. andere Funktionen aus.

Bei der Übergabe von Parametern an eine Funktion ist darauf zu achten, dass diese jeweils durch Kommata voneinander getrennt werden. Für die Namen der Parameter gelten die Regeln für die Namensbildung in JavaScript.

```
<SCRIPT type="text/JavaScript">
<!--
 function Ausrichter(Richtung)
{
 document.getElementById("TextAbsatz").align = Richtung;
}
//-->
</SCRIPT>
```

Im aufgeführten Beispiel, und auch hier soll nicht auf die Syntax der Funktion eingegangen werden, wird der Funktion `Ausrichter()` der Parameter `Richtung` übergeben. Wie sich diese Funktion aufrufen lässt, erfahren Sie im folgenden Abschnitt.

4.6.4 Funktionen mit Parametern aufrufen

vergleichbare Syntax

Der Aufruf von Funktionen mit Parametern unterscheidet sich nicht grundlegend von denen ohne Parameter. Dennoch ist auf einige syntaktische Feinheiten zu achten. Diese sollen anhand der nachstehenden Syntax veranschaulicht werden. Hier wird der Funktion `Ausrichter()` der Parameter `Richtung` übergeben. Die Funktion gestattet es, einen durch **<P>** gekennzeichneten und mit der ID `TextAbsatz` eindeutig referenzierten Textabsatz auszurichten. Hierzu wird innerhalb der Funktion über **getElementByID** auf das Element `Textabsatz` innerhalb des Dateikörpers zugegriffen. Die Eigenschaft **align** greift auf die horizontale Ausrichtung des Textabsatzes zu. Wie die Ausrichtung erfolgen soll, wird über den Parameter `Ausrichtung` festgelegt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
 function Ausrichter(Richtung)
{
 document.getElementById("TextAbsatz").align = Richtung;
}
//-->
</SCRIPT>
</HEAD
</BODY>
<P id="TextAbsatz">Richte Mich!</P>
<A href="javascript:Ausrichter('right')">rechts</A><BR>
<A href="javascript:Ausrichter('center')"> zentriert</A><BR>
<A href="javascript:Ausrichter('left')">links</A>
</BODY>
</HTML>
```

Listing 4.10: Funktion `Ausrichter()` mit dem Parameter `Richtung`

Innerhalb des **<A>**-Tags werden dem **href**-Attribut das Schlüsselwort **javascript** sowie dem sich hieran anschließenden Doppelpunkt der Funktionsname `Ausrichter()` zugewiesen. Der Parameter wird innerhalb der Klammern in Hochkommata notiert. Die Werte des Parameters bestimmen in diesem Beispiel die Ausrichtung des Textabsatzes. Zu diesem Zweck werden als Parameter die Werte **right**, **center** und **left** übergeben. Der Vorteil der Parameterübergabe in diesem Beispiel liegt auf der Hand. Der Textabsatz kann auf drei verschiedene Varianten ausgerichtet werden und dies unter Zuhilfenahme lediglich einer einzigen Funktion.

Abbildung 4.4: Die Seite nach dem Laden

Abbildung 4.5: Der Textabsatz ist nun zentriert.

4.6.5 Funktionen mit Wertrückgabe

boolesche Werte, Zahlen oder Strings

Um Werte aus Funktionen und Event-Handlern zurückzuliefern, wird das Schlüsselwort **`return`** verwendet. Funktionen können als Rückgabewerte boolesche Werte, Zahlen oder Strings liefern. Anhand zweier Beispiele soll der Umgang mit Rückgabewerten und deren Verwendungszweck veranschaulicht werden. Die erste Syntax zeigt, wie der Parameter einer Funktion mit einem String belegt ist, wenn diese Funktion aufgerufen wird. Im Ergebnis schreibt die Funktion das Wort `Wertrückgabe` in Großbuchstaben in das Dokument.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function schreibeWort (Zeichenkette)
{
Zeichen = Zeichenkette.toUpperCase();
return Zeichen;
}
var Wert =schreibeWort("Wertrückgabe")
```

```
document.write(Wert);
//-->
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.11: Der Inhalt der `Zeichen`-Variablen als Rückgabewert

Der Variablen `Wert` werden der Funktionsname `schreibeWort()` sowie der String `Wertrückgabe` übergeben. Innerhalb der Funktion wird die Variable `Zeichen` deklariert. Dieser werden der Parameter `Zeichenkette` sowie die Methode **`toUpperCase()`** zugewiesen. Als Rückgabewert liefert die `schreibeWort()`-Funktion den Inhalt der Variablen `Zeichen`.

Besonders häufig wird die **`return`**-Anweisung im Zusammenhang mit Formularen und Event-Handlern eingesetzt. So lässt sich hierdurch komfortabel überprüfen, ob das Formular tatsächlich abgeschickt werden soll. Dieses Vorgehen eignet sich somit vorzüglich für die Verwendung innerhalb von Programmen für die Formularvalidierung.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Sender()
{
var Wert = window.confirm("Empfänger " + document.Formular.action);
return Wert;
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular" action=mailto:name@provider.de
onsubmit="return Sender()">
<INPUT type="text" size="20" name="Name">
<INPUT type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 4.12: Nur wenn der OK-Button angeklickt wird, kann das Formular versendet werden.

Innerhalb des Dateikörpers befinden sich ein Formular, ein einzeiliges Eingabefeld sowie ein **`submit`**-Button. Das Anklicken des Buttons löst die Funktion `Sender()` aus. Hierzu wird der Event-Handler **`onsubmit`** einge-

setzt. Dieser Event-Handler ermittelt den Rückgabewert der Funktion `Sender()`. Ist dieser **`true`**, wird das Formular versandt. Ist er **`false`**, wird es nicht abgeschickt. Innerhalb der `Sender()`-Funktion wird der Variablen `Wert` die **`confirm`**-Eigenschaft des **`window`**-Objekts zugewiesen. Jedes **`confirm()`**-Fenster besteht aus einem OK- und einen Abbrechen-Button. Beim Anklicken des OK-Buttons wird der Wert **`true`** und beim Anklicken des Abbrechen-Buttons der Wert **`false`** zurückgeliefert. Der hier ermittelte Wert wird in der Variablen `Wert` gespeichert und von der Funktion selbst zurückgegeben. Das Formular wird also nur dann abgesendet, wenn der OK-Button angeklickt wurde.

4.6.6 Funktionen als Objekte

Funktionen können als Objekte definiert werden. Nähere Informationen zum Umgang mit Objekten erhalten Sie ab Kapitel 4.7. Durch die hier vorgestellte Variante wird es möglich, Funktionen zur Laufzeit des Programms zu definieren. Der grundlegende Aufbau einer Funktion, die als Objekt definiert wird, stellt sich folgendermaßen dar:

```
var name = new function (
[parameter_1] [, parameter_2] […,parameter_n,]
Programm-Anweisungen)
```

Hinter dem Schlüsselwort **`var`** wird der Name der Variablen definiert, in welcher der Rückgabewert der Funktion gespeichert werden soll. Dieser muss sich an den Konventionen für die Namensvergabe in JavaScript orientieren. Das Funktionsobjekt wird über die Schlüsselwörter **`new`** und **`function`** erzeugt. Innerhalb der Klammern werden die Parameter angegeben. Diese werden von der Funktion bis zum vorletzten Parameter interpretiert. Der letzte Parameter wird von JavaScript automatisch als Programm-Anweisung gedeutet. Sehen wir uns zum besseren Verständnis die nachfolgende Beispielsyntax an. Hier werden sowohl die Funktionsdefinition wie auch der Funktionsaufruf veranschaulicht. Im Ergebnis kann die Hintergrundfarbe der Datei modifiziert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Hintergrund =
new Function("FarbWert","document.bgColor=FarbWert;");
// -->
</SCRIPT>
</HEAD>
</BODY>
</BODY>
```

```
<A href="javascript:Hintergrund('#FFF000')">Hintergrundfarbe</A>
</HTML>
```

Listing 4.13: Eine Funktion als Objekt

Der Rückgabewert der Funktion wird innerhalb der `Hintergrund`-Variablen gespeichert. Hieran schließen sich das Gleichheitszeichen sowie die beiden Schlüsselwörter **`new`** und **`function`** an. Innerhalb der Klammer wird der Parameter `FarbWert` definiert. Die Programm-Anweisungen greifen auf die **`bgColor`**-Eigenschaft des **`document`**-Objekts zu. Als **`bgColor`**-Wert wird der Parameter `FarbWert` übergeben. Aufgerufen wird die Funktion durch Anklicken des Verweises, der sich innerhalb des Dateikörpers befindet. Dies geschieht über die Notation der Variablen `Hintergrund`. In dieser wird der Rückgabewert der Funktion gespeichert. Als Funktionsparameter wird ein hexadezimaler Farbwert verwendet.

4.6.7 Integrierte Funktionen

jederzeit aufrufbar

In JavaScript existieren fest eingebaute Funktionen, bei denen es sich nicht um Methoden handelt, die fest an ein Objekt gebunden sind. Anders als bei selbst definierten Funktionen sind diese in JavaScript bereits vordefiniert. Dies bedeutet, dass sich diese Funktionen jederzeit aufrufen lassen. Die Kenntnis von eingebauten Funktionen ermöglicht es dem Programmierer, schnell auf bestimmte Anforderungen reagieren zu können. So lassen sich beispielsweise einige dieser Funktionen trefflich für eine Plausibilitätskontrolle in Formularen einsetzen. Mehr zu den entsprechenden Einsatzgebieten erfahren Sie im Laufe dieses Abschnitts. JavaScript kennt die folgenden eingebauten Funktionen:

- **`DecodeURI()`** – dekodiert einen kodierten URI
- **`DecodeURIComponent()`** – dekodiert einen kodierten URI
- **`EncodeURI()`** – kodiert einen URI
- **`EncodeURIComnponent()`** – kodiert einen URI
- **`escape()`** – wandelt ASCII-Zeichen in Zahlen um
- **`eval()`** – dient der Berechnung eines Ausdrucks
- **`getClass`** – liefert die dazugehörige Java-Klasse zurück
- **`isFinite()`** – überprüft die Gültigkeit einer Zahl nach JavaScript
- **`isNaN()`** – dient der Überprüfung, ob es sich bei dem Wert um einen numerischen handelt
- **`parseFloat()`** – wandelt einen Wert in eine Kommazahl um
- **`parseInt()`** – wandelt einen Wert in eine ganze Zahl um
- **`number()`** – dient der Überprüfung, ob es sich bei dem Wert um einen numerischen handelt

- **string()** – konvertiert den Inhalt eines Objekts in eine Zeichenkette
- **unescape()** – wandelt Zahlen in ASCII-Zeichen um

Vor allem im Hinblick auf ältere Betriebssysteme und veraltete Browser bergen einige der aufgeführten Funktionen ein gewisses Maß an Fehleranfälligkeit in sich, die jedoch, so sie nicht auch in aktuellen Betriebssystem- oder Browserversionen auftauchen, hier keine Erwähnung finden sollen. Auf den folgenden Seiten werden die Bedeutung und der Einsatz von eingebauten Funktionen anhand von Beispielen veranschaulicht.

DecodeURI()

URIs dekodieren

Die **decodeURI()**-Funktion dekodiert einen durch **encodeURI()** kodierten URI. Weiterführende Informationen, wie und warum URIs kodiert werden, finden Sie im Abschnitt über **encodeURI()** ab Seite 104. Im folgenden Beispiel befinden sich drei einzeilige Eingabefelder sowie zwei Formular-Buttons. Der URI, welcher in das erste Feld eingetragen wird, wird als kodierter URI im zweiten Formularfeld angezeigt. Im dritten Eingabefeld wird der kodierte URI anschließend wieder dekodiert angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Kodieren()
{
var Eingabe = document.Formular.Eingabe.value;
var Adresse = encodeURI(Eingabe);
document.Formular.Ausgabe.value = Adresse;
}
function Dekodieren()
{
var KodierteAdresse = document.Formular.Ausgabe.value;
var DekodierteAdresse = decodeURI(KodierteAdresse);
document.Formular.Ausgabe_deko.value = DekodierteAdresse;
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="">
<INPUT type="text" name="Ausgabe" value="">
<INPUT type="text" name="Ausgabe_deko" value="">
<INPUT type="button" onclick="Kodieren()" value="kodieren">
```

```
<INPUT type="button" onclick="Dekodieren()" value="dekodieren">
</FORM>
</BODY>
</HTML>
```

Listing 4.14: Kodieren und dekodieren

Um den URI zu kodieren, wird die Funktion `Kodieren()` verwendet. Innerhalb dieser Funktion wird zunächst die Variable `Eingabe` deklariert. Als deren Wert wird der Wert des Eingabelfeldes mit dem Namen `Eingabe` definiert. Im nächsten Schritt wird der Variablen `Adresse` die **`encodeURI()`**-Funktion zugewiesen. Der hier ermittelte Wert wird anschließend innerhalb des Eingabefeldes mit dem Namen `Ausgabe` angezeigt.

Die Funktion **`Dekodieren()`** dient der Dekodierung des innerhalb des Eingabefeldes mit dem Namen `Ausgabe` angezeigten Wertes. Innerhalb dieser Funktion wird im ersten Schritt die kodierte Variable deklariert. Dieser wird der Wert des Eingabefeldes `Ausgabe` zugewiesen. Die Variable `Adresse_deko` erhält als Wert die Funktion **`decodeURI()`**. Abschließend wird der Wert des Eingabefeldes `Ausgabe_deko` mit dem ermittelten Wert der Variablen `Adresse_deko()` gleichgesetzt.

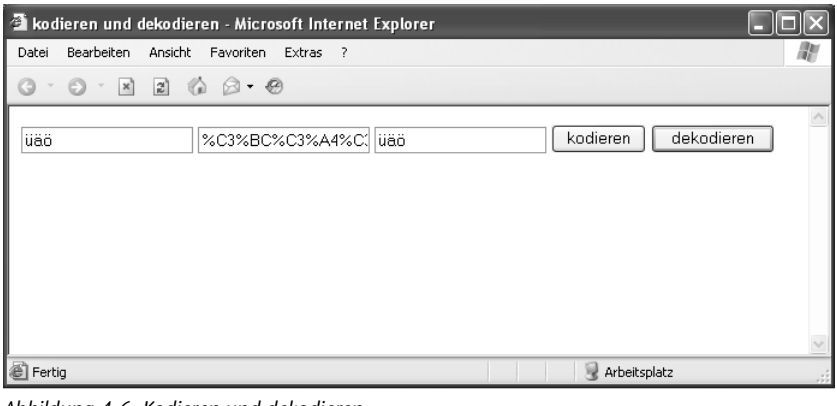


Abbildung 4.6: Kodieren und dekodieren

DecodeURIComponent()

URIs dekodieren

Mittels der **`DecodeURIComponent()`**-Funktion können kodierte URIs dekodiert werden. Es handelt sich hierbei um eine ähnliche Anwendung wie bei **`decodeURI()`**. Der Unterschied besteht lediglich darin, dass durch die **`DecodeURIComponent()`**-Funktion nur solche URIs dekodiert werden sollten, die durch die Funktion **`encodeURIComponent()`** kodiert wurden. Die nachstehende Syntax beschreibt, wie sich ein URI zunächst kodieren und anschließend wieder dekodieren lässt. Beide URIs, also sowohl der kodierte wie auch der dekodierte, werden nacheinander in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Dekodierer()
{
var URL = "http://www.server.com?script.pl? input = 34";
var URLAusgabe = encodeURIComponent(URL);
alert("kodiert: " + URLAusgabe);
URLDekodieren = decodeURIComponent(URL);
alert("dekodiert:" + URLDekodieren);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:Dekodierer ()">
http://www.server.com?script.pl? input = 34</A>
</BODY>
</HTML>
```

Listing 4.15: URI dekodieren

Durch Anklicken des Hyperlinks wird die Funktion `Dekodierer()` aufgerufen. Innerhalb dieser Funktion wird die Variable `URL` deklariert. Als Wert wird dieser ein URI zugewiesen. Anschließend wird der URI durch die Funktion **`encodeURIComponent()`** kodiert und der ermittelte Wert über die **`alert()`**-Methode ausgegeben. Der kodierte URI wird anschließend der Funktion **`decodeURIComponent()`** übergeben und von dieser dekodiert. Auch der hier ermittelte Wert wird mittels der **`alert()`**-Methode ausgegeben.

EncodeURI()

URI's kodieren

Die **`EncodeURI()`**-Funktion ermöglicht das Kodieren eines URIs, so dass alle vorkommenden Sonderzeichen in ASCII-Zeichen umgewandelt werden. Besonders wichtig ist dies vor dem Hintergrund von CGI-Scripts und der Parameterübergabe, die ja bekanntlich häufig aus Sonderzeichen, Leerzeichen usw. bestehen. Um HTML-4-konform zu arbeiten, muss jeder vorkommende URI ohne Sonderzeichen auskommen. Die **`EncodeURI()`**-Funktion kodiert alle Zeichen, außer den folgenden:

Ausnahmen

- Buchstaben von A bis Z
- Buchstaben von a bis z
- @ & _ - . * ' ( ) , = + / ? : $! ~

Im folgenden Beispiel wird ein URI kodiert und der kodierte Wert innerhalb eines Meldungsfensters angezeigt. Das Beispiel soll vor allem verdeutlichen, dass auch Umlaute kodiert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Kodieren()
{
var Zeiger = encodeURI("http://www.prodiver.com/möchte gern/ hinüber");
alert(Zeiger);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:Kodieren()">kodierte Adresse anzeigen</A>
</BODY>
</HTML>
```

Listing 4.16: Kodieren eines URIs

Die Funktion **codi()** wird durch Anklicken des Hyperlinks ausgelöst. Innerhalb der Funktion wird zunächst die Variable `Zeiger` deklariert. Dieser wird als Wert die **encodeURI()**-Funktion mit einem URI übergegeben. Der URI enthält Umlaute, die laut HTML 4 kodiert werden müssen. Abschließend wird der kodierte URI mittels der **alert()**-Methode ausgegeben.

Abbildung 4.7: Der kodierte URI in einem Meldungsfenster

EncodeURIComponent()

Sonderzeichen in ASCII-Zeichen umwandeln

Die **EncodeURIComponent()**-Funktion kodiert einen URI so, dass alle Sonderzeichen in ASCII-Zeichen umgewandelt werden. Interessant ist dies vor allem vor dem Hintergrund von HTML-4-konformen Seiten und der Verwendung von CGI-Scripts mit Parameterübergabe. Die HTML-4-Spezifikation sieht vor, dass URI's keine Sonderzeichen enthalten dürfen, sondern diese als kodierte Zeichenketten anzugeben sind. Die **EncodeURIComponent()**-Funktion kodiert alle Zeichen außer den folgenden:

- Buchtaben von A bis Z
- Buchstaben von a bis z
- _ - . * ' ( ) ! ~

Anhand der nachstehenden Syntax wird gezeigt, wie ein Hyperlink so kodiert wird, dass es sich anschließend um einen HTML-4-konformen Verweis handelt. Der kodierte Hyperlink wird innerhalb eines Meldungsfensters angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Kodieren()
{
var Zeiger =
encodeURIComponent("http://www.prodiver.com/möchte
gern?new.pl = nö");
alert(Zeiger);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:Kodieren()">kodierte Adresse anzeigen</A>
</BODY>
</HTML>
```

Listing 4.17: URIs kodieren

Die Funktion `Kodieren()` wird durch Anklicken des Verweises ausgelöst. Der Variablen `Zeiger` wird die **`encodeURIComponent()`**-Funktion mit einem entsprechenden URI zugewiesen. Dieser URI wird anschließend mittels der **`alert()`**-Methode in kodierter Form ausgegeben.

Escape()

Sonderzeichen in ASCII-Zeichen umwandeln

Die **`Escape()`**-Funktion wandelt Sonder- und Steuerzeichen in ASCII-Zahlenwerte um. Vor jedes Zeichen wird zudem automatisch das `%`-Zeichen gesetzt. So wird beispielsweise aus dem Umlaut *ä* die Zeichenfolge *%E4*. Trifft die **`Escape()`**-Funktion auf andere Zeichen außer Sonder- und Steuerzeichen, werden diese nicht verändert. Die nachstehende Syntax zeigt, wie Sonderzeichen, die sich innerhalb eines Eingabefeldes befinden, durch die **`Escape()`**-Funktion in Zahlenwerte umgewandelt und in einem Meldungsfenster ausgegeben werden können.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Wandler()
{
var Inhalt = document.Formular.Eingabe.value;
alert(escape(Inhalt));
}
//-->
</SCRIPT
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="äüö">
<INPUT type="button" onclick="Wandler()" value="anzeigen">
</FORM>
</BODY>
</HTML>
```

Listing 4.18: Sonderzeichen und Leerzeichen umwandeln

Der Event-Handler **onclick** löst die Funktion `Wandler()` aus. Zu Beginn der Funktion wird die Variable `Inhalt` deklariert. Dieser wird der Wert des einzeiligen Eingabefeldes `Eingabe` zugewiesen. Abschließend wird auf den hier ermittelten Wert mittels der **escape()**-Funktion zugegriffen und das Ergebnis über die **alert()**-Methode ausgegeben.

Eval()

ein kleiner Rechner

Mit der **eval()**-Funktion steht ein einfach zu handhabendes Werkzeug zur Verfügung, mit dessen Hilfe sich mathematische Berechnungen realisieren lassen. Die **eval()**-Funktion erhält als Parameter eine Zeichenkette mit einem JavaScript-Ausdruck und wertet diesen aus. Diese Funktion eignet sich beispielsweise für das Bereitstellen eines Formulars, mit dessen Hilfe sich mathematische Berechnungen durchführen lassen. Eine solche Anwendung beschreibt das folgende Beispiel. Hier werden ein einzeiliges Textfeld sowie ein herkömmlicher HTML-Button definiert. Diesem Button wird die **eval()**-Funktion zugewiesen. Gibt der Nutzer in das Textfeld einen mathematischen Ausdruck ein und benutzt er den Button, wird das Ergebnis der Berechnung in dem Eingabefeld ausgegeben. Die gezeigte Syntax birgt allerdings das Problem der nur unzureichenden Fehlerbehandlung in sich. Dies lässt einen praktischen Einsatz ohne zusätzliche Modifikation eher unwahrscheinlich erscheinen. Denn gibt der Nutzer innerhalb des Textfeldes keinen mathematischen Ausdruck ein, wird eine JavaScript-Fehlermeldung ausgegeben.

Umgehen lässt sich dies allerdings auf einfache Weise. Wie genau dies geschehen kann, wird im nachfolgenden **parseFloat()**-Abschnitt beschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Ergebnis()
{
Rechner = document.Formular.Eingabe.value;
document.Formular.Eingabe.value = (eval(Rechner));
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT name="Eingabe" value="34+89">
<INPUT type="button" value="Berechnen" onclick="Ergebnis()">
</FORM>
</BODY>
</HTML>
```

Listing 4.19: Ein kleiner Rechner

JavaScript ausführen

Neben rein mathematischen Anwendungen lässt sich die **eval()**-Funktion auch für das Ausführen von JavaScript-Anweisungen nutzen. Ist dies gewünscht, muss als Parameter der **eval()**-Funktion lediglich ein regulärer JavaScript-Ausdruck notiert werden.

GetClass()

Java-Klasse ermitteln

Mittels der **GetClass()**-Funktion kann die zu einem angegebenen **applet**- oder **embed**-Objekt gehörende Java-Klasse ermittelt werden. Als Rückgabewert wird der Name der Klassendatei geliefert. Als Parameter wird das Objekt erwartet. Ein Beispiel mit einem integrierten Java-Applet zeigt die folgende Syntax:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeKlasse()
{
var Klasse = getClass(document.applets[0])
document.write(Klasse);
```

```
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<APPLET code="../myprojects/paint.class" codebase="../Netzwerkumgebung"
width="300" height="100"></APPLET>
<A href="javascript:zeigeKlasse()">Die verwendete Klasse ist:</A>
</BODY>
</HTML>
```

Listing 4.20: Die Java-Klasse wird in das Dokument geschrieben.

Durch Anklicken des Verweises wird die Funktion `zeigeKlasse()` aufgerufen. Der Variablen `Klasse` wird die **`getClass()`**-Funktion zugewiesen. Diese erhält als Parameter das Objekt. Hier ist dies das erste innerhalb des Dateikörpers vorkommende Java-Applet. Der Rückgabewert, also der Klassenname, wird durch die **`write()`**-Methode dynamisch in das Dokument geschrieben.

IsFinite()

Gültigkeitsbereich überprüfen

Die **`IsFinite()`**-Funktion überprüft, ob sich eine Zahl im JavaScript-Gültigkeitsbereich befindet, also ob die Zahl von JavaScript verarbeitet werden kann. Die maximale von JavaScript zu verarbeitende Zahl lässt sich über **`MAX_VALUE`** und die kleinste zu verarbeitende Zahl über **`MIN_VALUE`** des **`Number`**-Objekts ermitteln. Als Rückgabewert liefert die **`IsFinite()`**-Funktion einen der boolschen Werte **`true`** und **`false`**.

mögliche Werte

- **`true`** – Die Zahl liegt im Gültigkeitsbereich.
- **`false`** – Die Zahl liegt nicht im Gültigkeitsbereich.

Der **`IsFinite()`**-Funktion können sowohl ganze Zahlen wie auch Kommazahlen (allerdings mit Dezimalpunkt notiert) zugewiesen werden. Handelt es sich bei den übergebenen Zeichen nicht um eine Ziffer, sondern beispielsweise um einen Buchstaben, wird nicht etwa **`NaN`**, sondern ebenfalls **`false`** zurückgeliefert. Im folgenden Beispiel befindet sich ein einzeiliges Eingabefeld. Ein hierin notierter Wert wird daraufhin überprüft, ob er sich im JavaScript-Gültigkeitsbereich befindet. Das entsprechende Ergebnis wird in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Pruefer()
{
```

```
var Ziffer = document.Formular.eingabe.value;
if (isFinite(Ziffer) == true)
alert("ja")
else
alert("nein")
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="text" name="eingabe" value="">
<A href="javascript:Pruefer()">Ist es m&ouml;glich</A>
</FORM>
</BODY>
</HTML>
```

Listing 4.21: Ist die Zahl im Gültigkeitsbereich von JavaScript?

Die `Pruefer()`-Funktion wird durch Anklicken des Verweises ausgelöst. Innerhalb dieser Funktion wird zunächst die Variable `Ziffer` deklariert. Dieser wird der Wert des einzeiligen Eingabefeldes zugewiesen. Anschließend wird in Form einer **`if`**-Abfrage überprüft, ob sich der Wert innerhalb des JavaScript-Gültigkeitbereichs befindet. Die jeweilige Meldung wird über die **`alert()`**-Methode ausgegeben.

IsNAN()

Handelt es sich um eine Zahl?

Die Funktion **`isNaN()`** überprüft, ob es sich bei dem übergebenen Wert um eine Zahl handelt. Diese Funktion gibt als Ergebnis die boolschen Operatoren **`true (wenn es sich um eine Zahl handelt)`** oder **`false`** (wenn es sich um keine Zahl handelt) zurück. Der Nachteil dieser Funktion ist, dass der Netscape Navigator in der Version 2 diese nur in der Unix-Version beherrscht. Soll die entsprechende Anwendung also auch auf diesem Netscape-Produkt korrekt funktionieren, muss eine eigene Funktion geschrieben werden. Im Normalfall kann **`isNaN()`** aber dennoch in den verschiedensten Anwendungsbereichen eingesetzt werden. So lassen sich mit dieser Funktion beispielsweise auf einfachem Wege Plausibilitätskontrollen innerhalb von Formularen durchführen. Und eben einen solchen Fall beschreibt die folgende Beispiel-Syntax:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function pruefe(Zahl)
{
```

```
if(isNaN(Zahl) == true)
{
alert("Das ist keine Zahl");
return false;
}
else return true;
}
</SCRIPT>
</HEAD>
</BODY>
<FORM>
<INPUT type="text" name="wert" size=15>
<INPUT type="button" value="test" onclick="pruefe(this.form.wert.value)">
</FORM>
</BODY>
</HTML>
```

Listing 4.22: Handelt es sich um eine Zahl?

In dem gezeigten Formular werden ein einzeiliges Eingabefeld sowie ein HTML-Button definiert. Gibt der Nutzer innerhalb des Feldes einen anderen Wert als einen numerischen ein, wird ein Meldungsfenster mit der Botschaft `Das ist keine Zahl` ausgegeben. Wobei JavaScript bei der Bewertung, ob es sich um eine gültige Zahl handelt, zwischen einer Ganzzahl und einer Kommazahl keine Unterscheidung vornimmt.

ParseFloat()

Zeichenketten in Zahlenwerte umwandeln

Die **`ParseFloat()`**-Funktion wandelt eine Zeichenkette in einen Zahlenwert um. Dies gilt jedoch nicht, wenn die Zeichenkette mit einem anderen Zeichen als einer Ziffer beginnt. Ist dies der Fall, wird **`NaN`** zurückgeliefert. Einige Beispiele sollen zeigen, welche Werte laut **`ParseFloat()`** als gültig betrachtet werden.

Eingabe	ParseFloat()
200	200
200.95	200.95
200,78	200
34abc56	34
A45	NaN

Tabelle 4.12: Beispiele für `parseFloat()`-Auswirkungen

Die aufgeführte Tabelle konnte selbstverständlich nur einen kleinen Teil möglicher Werte abdecken. Um diesen „Missstand“ beseitigen zu können, steht die nachstehende Syntax zur Verfügung. Hier können

numerische Werte in ein Eingabefeld eingegeben und berechnet werden. Der ermittelte Wert kann anschließend anhand von **ParseFloat()** überprüft und in einem Meldungsfenster angezeigt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Berechner()
{
Rechner = document.Formular.Eingabe.value;
document.Formular.Eingabe.value = (eval(Rechner));
}
function Ueberpruefer()
{
Wert = document.Formular.Eingabe.value;
alert(parseFloat(Wert));
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT size="30" name="Eingabe" value="34">
<INPUT type="button" value="Berechnen" onclick="Berechner()">
<INPUT type="button" value="wandler" onclick="Ueberpruefer()" >
</FORM>
</BODY>
</HTML>
```

Listing 4.23: Der berechnete Wert wird in einen Zahlenwert umgewandelt.

Die Funktion `Berechner()` wird durch den Event-Handler **onclick** aufgerufen. Hierin wird zunächst die Variable `Rechner` deklariert. Als Wert wird dieser der Wert des einzeiligen Eingabefeldes zugewiesen. Anschließend wird der Wert des Eingabefeldes mittels der **Eval()**-Funktion berechnet. Der ermittelte Wert wird im Eingabefeld angezeigt.

Die zweite Funktion `Ueberpruefer()` dient der Umwandlung eingegebener Zeichen in **ParseFloat()**-Werte. Zum Aufruf der Funktion dient auch hier der Event-Handler **onclick**. Innerhalb der Funktion wird zunächst die Variable `Wert` deklariert. Diese bekommt als Wert den Wert des einzeiligen Eingabefeldes zugewiesen. Im nächsten Schritt wird der hier ermittelte Wert der **ParseFloat()**-Funktion übergeben. Das Ergebnis wird mittels der **alert()**-Methode ausgegeben.

parseInt()

Zeichenkette in eine Ganzzahl umwandeln

Über die Funktion **parseInt()** kann eine übergebene Zeichenkette in eine Ganzzahl umgewandelt werden. Sollte die Zeichenkette mit einem anderen Zeichen als einer Ziffer beginnen, wird **NaN** zurückgeliefert. Sollte jedoch das erste Zeichen eine Zahl, die folgenden jedoch Buchstaben sein, so wird die Zahl bis zu dem Buchstaben interpretiert. Die folgende Tabelle zeigt einige denkbare Zeichenketten und deren Interpretation durch die **parseInt()**-Funktion.

Eingabe	ParseInt()
200	200
200.95	200
200,78	200
34abc56	34
A45	**NaN**

Tabelle 4.13: Beispiele für parseInt()-Auswirkungen

Anhand der nachstehenden Syntax wird es Ihnen möglich, selbst zu überprüfen, welche Werte die **parseInt()**-Funktion zurückliefert. Innerhalb des Beispiels befindet sich ein einzeiliges Eingabefeld. Hierin können Zeichenketten notiert werden. Diese werden anschließend mittels der **parseInt()**-Funktion überprüft und die zurückgelieferte Zeichenkette wird in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Ganzzahl()
{
var Inhalt = document.Formular.Eingabe.value;
alert(parseInt(Inhalt));
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="345">
<INPUT type="button" onclick="Ganzzahl()" value="zeige Wert">
</FORM>
</BODY>
</HTML>
```

Listing 4.24: Die Ausgabe von Ganzzahlen

Die Funktion `Ganzzahl()` wird durch Anklicken des Formular-Buttons ausgelöst. Innerhalb der Funktion wird zunächst die Variable `Inhalt` deklariert. Als Wert erhält diese den Wert des einzeiligen Eingabefeldes. Dieser Wert wird der Funktion **`parseInt()`** als Parameter übergeben und mittels der **`alert()`**-Methode ausgegeben.

Number()

Wert in eine Zahl umwandeln

Die **`Number()`**-Funktion wandelt einen übergegebenen Wert in eine Zahl um. Sollte der übergebene Wert sich nicht als Zahlenwert interpretieren lassen, wird **`NaN`** zurückgeliefert. Im folgenden Beispiel soll dies anhand zweier Werte veranschaulicht werden. Bei den übergebenen Zeichenketten handelt es sich einmal um einen und einmal um keinen Zahlenwert. Beide Wert werden auf ihre **`Number()`**-Gültigkeit hin überprüft und das Ergebnis wird anschließend in jeweils einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
</HEAD>
</BODY>
<SCRIPT type="text/JavaScript">
<!--
 var Inhalt = "456.67";
 var Inhalt2 = "Keine Zahl";
 alert(Number(Inhalt));
 alert(Number(Inhalt2));
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 4.25: Beide Werte werden in einem Meldungsfenster ausgegeben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgeführt. Zunächst werden zwei Variablendeklarationen vorgenommen, wobei der Variablen `Inhalt` eine Zahl und der Variablen `Inhalt2` eine Zeichenkette zugewiesen wird. Anschließend werden die beiden Variablen jeweils einer **`Number()`**-Funktion als Parameter übergeben und die entsprechenden Werte über die **`alert()`**-Methode ausgegeben. Der erste Wert kann interpretiert werden. Hier erscheint als Ausgabe `456.67`. Da es sich bei dem zweiten Wert um keine gültige Zahl handelt, wird hier **`NaN`** angezeigt.

String()

Objektinhalt in eine Zeichenkette umwandeln

Die **String()**-Funktion ermöglicht es, den Inhalt eines Objekts in eine Zeichenkette zu konvertieren. Als Ergebnis liefert diese Funktion wiederum eine Zeichenkette zurück. Der Einsatz von **String()** ist vor allem immer dann sinnvoll, wenn innerhalb eines Scripts mit dem **Date()**-Objekt gearbeitet wird. Einen solchen Fall beschreibt die folgende Syntax, als deren Ergebnis das aktuelle Datum innerhalb eines Meldungsfensters ausgegeben wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Zeit()
{
 var Tag = new Date ();
 alert(String(Tag));
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<INPUT type="button" onclick="Zeit()" value="zeige Zeit">
</BODY>
</HTML>
```

Listing 4.26: Das Datum wird als Zeichenkette ausgegeben.

Durch das Anklicken des innerhalb des Dateikörpers befindlichen Formular-Buttons wird die Funktion `Zeit()` aufgerufen. In dieser wird innerhalb der Variablen `Tag` ein neues Datumsobjekt gespeichert. Dieses wiederum wird über die Anweisung **String(**`Tag`**)** in eine Zeichenkette konvertiert und anschließend in einem **alert()**-Fenster ausgegeben.

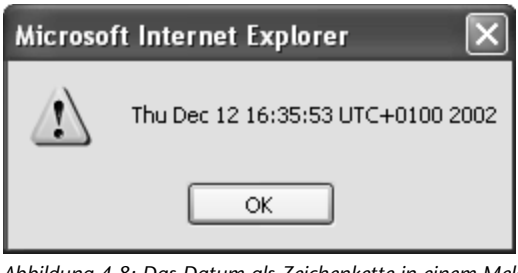


Abbildung 4.8: Das Datum als Zeichenkette in einem Meldungsfenster

Unescape()

Zeichen in ASCII-Zeichen umwandeln

Die **Unescape()**-Funktion wandelt alle übergebenen Zeichen in ASCII-Zeichen um. Jedem Zeichen einer zu übergebenden Zeichenkette muss ein **%**-Zeichen vorangestellt werden. Das Zeichen selbst muss aus dem hexadezimalen Wert des Zeichens aus der ASCII-Tabelle bestehen. So wird beispielsweise aus dem Zeichen *%6E* der **Unescape()**-Wert *ü*. Im folgenden Beispiel befinden sich drei Eingabefelder. Innerhalb des ersten Feldes wird standardmäßig die Zeichenfolge äüö angezeigt. Diese wird mittels der **escape()**-Funktion in ASCII-Zeichen umgewandelt. Der Wert des zweiten Feldes beträgt nun *%E4%FC%F6*. Im dritten Feld wird die übergebene Zeichenkette wieder wie im ersten Feld angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function wandelZeichen()
{
var Inhalt = document.Formular.Eingabe.value;
document.Formular.Ausgabe.value = escape(Inhalt);
}
function zurueckwandelZeichen()
{
var neuerInhalt = document.Formular.Ausgabe.value;
document.Formular.umwandeln.value = unescape(neuerInhalt);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="äüö">
<INPUT type="text" name="Ausgabe" value="">
<INPUT type="text" name="umwandeln" value="">
<INPUT type="button" onclick=" wandelZeichen();zurueckwandelZeichen()"
value="anzeigen">
</FORM>
</BODY>
</HTML>
```

Listing 4.27: Die eingegebenen Zeichen werden in ASCII-Zeichen umgewandelt.

Der Event-Handler **onclick** löst die beiden Funktionen `wandelZeichen()` und `zurueckwandelZeichen()` aus. In der `wandelZeichen()`-Funktion wird die Variable `Inhalt` deklariert und dieser der Wert des einzeiligen Eingabefeldes `Eingabe` zugewiesen. Als Nächstes wird dem Wert des Eingabefeldes

Ausgabe die Funktion **escape()** und als deren Parameter Inhalt übergeben. Somit wird innerhalb des Ausgabe-Feldes die Zeichenkette in ASCII-Zahlenwerten angezeigt.

Einem ähnlichen Prinzip folgt auch die zurueckwandelZeichen()-Funktion. Innerhalb dieser Funktion wird zunächst die Variable neuerInhalt deklariert. Dieser wird der Wert des einzeiligen Eingabefeldes Ausgabe zugewiesen. Im letzten Schritt wird der Wert des Eingabefeldes umwandeln mit der Funktion **unescape()** sowie deren Parameter neuerInhalt belegt.

4.7 Objekte

Eigenschaften und Methoden

Objekte sind ein wichtiger Bestandteil von JavaScript. Sie werden dazu verwendet, Daten strukturiert darzustellen. Objekte sind festgeschriebene Datenelemente, die Eigenschaften und Methoden besitzen können. In JavaScript steht eine Vielzahl von vordefinierten Objekten zur Verfügung. Gleichfalls ist es aber auch möglich, eigene Objekte zu definieren. Objekte können Unterobjekte eines anderen Objekts sein. Aus diesem Grund wurden alle JavaScript-Objekte in einer Objekthierarchie genauer angeordnet. Mehr zu dem Thema Objekthierarchie erfahren Sie in den folgenden Abschnitten. Neben solchen Objekten, die hierarchisch angeordnet sind, gibt es auch solche, die sich nicht explizit in der Objekthierarchie einordnen lassen. Beispielhaft hierfür sei das **Date**-Objekt genannt.

4.7.1 Vordefinierte Objekte

sofort einsetzbar

In JavaScript existiert bereits eine Vielzahl vordefinierter Objekte. Hierbei handelt es sich um solche Objekte, die Sie nicht selber definieren müssen. Sie können sofort eingesetzt werden. Durch vordefinierte Objekte können Sie beispielsweise auf die Elemente eines Formulars zugreifen. JavaScript stellt eine Fülle an vordefinierten Objekten bereit. Sie sollten in jedem Fall vor der Definition eigener Objekte überprüfen, ob für den gewünschten Zweck bereits vordefinierte existieren. Das folgende Beispiel zeigt die Verwendung des vordefinierten **document**-Objekts. Zusätzlich wird die Verwendung von Methoden und Eigenschaften gezeigt. Hierauf gehen wir im Lauf dieses Kapitels allerdings noch genauer ein.

```
<HTML>
<HEAD>
<TITLE>Herzlich Willkommen</TITLE>
</HEAD>
</BODY>
<H2>
```

```
<SCRIPT type="text/JavaScript">
<!--
 document.write(document.title);
//-->
</SCRIPT>
</H2>
</BODY>
</HTML>
```

Listing 4.28: Verwendung des vordefinierten **document***-Objekts*

Durch die gezeigte Syntax wird nach dem Laden der Seite der Dateititel `Herzlich Willkommen` dynamisch in das Dokument geschrieben. Um den Dateititel auszulesen, wird die **title**-Eigenschaft des **document**-Objekts verwendet. Das Schreiben in das Dokument übernimmt die **write()**-Methode des **document**-Objekts.

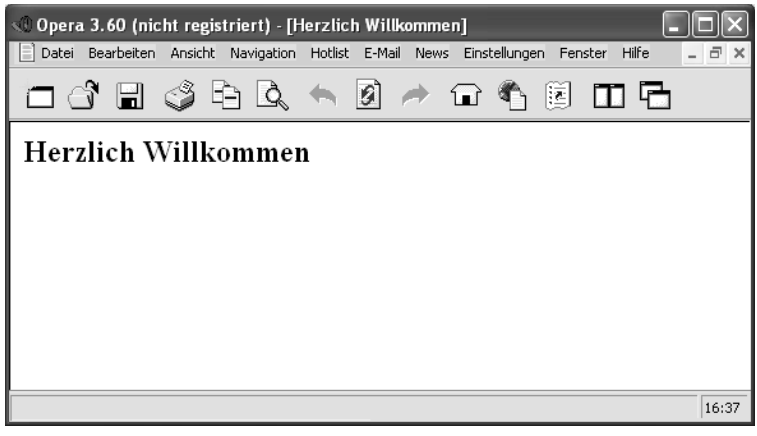


Abbildung 4.9: Der Datei-Titel wird im Dokument angezeigt.

4.7.2 Objekthierarchie von WWW-Seiten

***window** ist das oberste Objekt*

Vordefinierte Objekte sind in JavaScript hierarchisch angeordnet. Für Sie als Programmierer bedeutet das, dass Sie ohne Kenntnis der Objekthierarchie einer WWW-Seite nicht auf die gewünschten Objekte zugreifen können. HTML-Dokumente, die im Browser angezeigt werden, sind prinzipiell stets gleich aufgebaut. Dies erleichtert für Sie den Zugriff auf das gewünschte Objekt. Das oberste Objekt in der Objekthierarchie von JavaScript ist **window**. Mit diesem beginnt auch die aufgeführte Objekthierarchie. Noch ein Wort zum Stil: Das oberste Objekt einer Aufzählung zeigt stets das übergeordnete Objekt. Die eingerückt dargestellten Objekte sind Unterobjekte. So ist also beispielsweise **document** ein Unterobjekt von **window**.

- `window` – Fenster
  - `frames[ ]` – Array von Frames
  - `document` – Dokument im Anzeigebereich
  - `location` – URI
  - `history` – besuchte Seiten
  - `navigator` – Browserinformationen

Das `document`-Objekt, welches das aktuelle Dokument im Anzeigebereich spezifiziert, kann weitere Unterobjekte enthalten. Die nachstehende Auflistung zeigt diese möglichen Objekte.

- `document` – Dokument im Anzeigebereich
  - `anchors[ ]` – Verweisanker
  - `applets[ ]` – Java-Applets
  - `embeds[ ]` – Multimediaelemente
  - `forms[ ]` – Formulare
  - `images[ ]` – Grafiken
  - `layers[ ]` – Layer
  - `links[ ]` – Hyperlinks

Das `forms`-Objekt, welches Formulare spezifiziert, kann weitere Unterobjekte enthalten. Die nachstehende Auflistung zeigt diese möglichen Objekte.

- `forms[ ]` – Formulare
  - `elements[ ]` – Formularelemente eines Formulars
    - `button` – Button
    - `checkbox` – Checkbox
    - `fileupload` – Datei-Upload-Button
    - `hidden` – verstecktes Eingabefeld
    - `password` – Passwortfeld
    - `radio` – Radio-Button
    - `reset` – Button zum Löschen von Formulareingaben
    - `select` – Listenfeld
    - `submit` – Absende-Button
    - `text` – einzeiliges Eingabefeld
    - `textarea` – mehrzeiliges Eingabefeld

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

- `select` – Listenfeld
  - `options[ ]` – Einträge des Listenfeldes

Wie können Sie nun diese Objekthierarchie nutzen, um auf Objekte einer WWW-Seite zugreifen zu können? Nehmen wir an, dass eine HTML-Datei erstellt wird. Hierin ist ein Formular enthalten. In diesem befindet sich nun wiederum ein einzeiliges Eingabefeld. Das HTML-Konstrukt hierfür stellt sich folgendermaßen dar:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="E-Mail">
</FORM>
</BODY>
</HTML>
```

Listing 4.29: Das Formular wird für den JavaScript-Zugriff vorbereitet.

Indexnummer oder Name

Per JavaScript soll nun auf das Eingabefeld zugegriffen und dessen Wert ausgelesen werden. Es gibt zwei verschiedene Möglichkeiten, dies zu realisieren. Sie können entweder mit einer Indexnummer oder dem Namen auf das Formularelement zugreifen. Bei beiden Varianten gilt, dass die Objekthierarchie ordnungsgemäß durchlaufen werden muss. Führen wir uns die Objekthierarchie für dieses Beispiel vor Augen. Das oberste Objekt lautet **window**. Auf dessen Notation kann hier verzichtet werden. Das nächste Objekt bezieht sich auf das Anzeigefenster und ist **document**. Im nächsten Schritt wird auf das Formular zugegriffen. Hierzu wird das **forms**-Objekt verwendet. Über **elements** wird auf das Formularelement zugegriffen. Den Wert des Eingabefeldes ermittelt die **value**-Eigenschaft. Das hierzu passende JavaScript-Konstrukt stellt sich folgendermaßen dar:

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

Hier wird mit Indexnummern gearbeitet. Die Angabe **forms[0]** gibt an, dass sich die Anweisung auf das erste Formular in der Datei bezieht. Über **elements[0]** wird auf das erste Element in dem Formular zugegriffen. Beachten Sie, dass die Zählung in JavaScript bei 0 beginnt. Möchten Sie also beispielsweise auf das zweite Formular einer HTML-Datei zugreifen, müssen Sie **forms[1]** notieren. Bei **value** handelt es sich um eine Eigenschaft des **elements**-Objekts.

Nachteile bei Indexnummern

Die Verwendung von Indexnummern ist völlig korrekt. Dennoch kann dies zu Problemen bei der Pflege und Korrektur von Dateien führen. Nehmen wird folgendes Szenario an: Sie haben die zuvor gezeigte Seite erstellt und greifen über Indexnummern auf das Formularelement zu. Nach der Fertigstellung der Seite fällt Ihnen ein, dass vor dem vorhan-

denen ein zweites Formular eingefügt werden muss. Somit würde die Anweisung **forms[0]** nun nicht mehr auf das gewünschte, sondern auf das neu eingefügte Formular zugreifen. JavaScript-Fehler währen die Folge. Um dies zu umgehen, können Sie die einzelnen Objekte mit Namen ansprechen. Dieser Name ergibt sich aus der Zuweisung des **name**-Attributs zu den entsprechenden HTML-Tags.

```
document.Formular.Eingabe.value;
```

Mit dieser Anweisung erzielen Sie den gleichen Effekt wie bei der Verwendung von Indexnummern. Über `Formular` wird auf das Formular zugegriffen. Das einzeilige Eingabefeld wird über `Eingabe` angesprochen. Beachten Sie, dass dem **name**-Attribut des HTML-Tags exakt der gleiche Wert wie in der JavaScript-Anweisung zugewiesen werden muss. JavaScript unterscheidet zwischen Groß- und Kleinschreibung. Wurde dem HTML-Tag **<input>** der **name**-Wert `Eingabe` zugewiesen, würde die JavaScript-Anweisung `eingabe` eine Fehlermeldung erzeugen.

4.7.3 Eigene Objekte

eindeutige Namen

Neben der Möglichkeit, vordefinierte Objekte zu verwenden, können auch eigene Objekte erzeugt werden. Zwar stehen zahlreiche vordefinierte Objekte zur Verfügung, für viele Anwendungen reichen diese allerdings nicht aus. Objekten muss ein eindeutiger Name zugewiesen werden. Die Eigenschaften und Methoden werden, durch einen Punkt getrennt, hinter dem Objekt notiert. Nehmen wir an, dass dem Objekt `Autor` die Eigenschaft `Name` zugewiesen werden soll.

```
Autor.Name
```

Eigenschaften sind veränderbar. Dies verhält sich exakt so, wie dies auch im Umgang mit Variablen der Fall ist. Um einer Objekteigenschaft einen Wert zuzuweisen, werden das Gleichheitszeichen und anschließend der Wert notiert. Soll der Eigenschaft `Name` des `Autor`-Objekts der Wert `Hornby` zugewiesen werden, sieht dies folgendermaßen aus:

```
Autor.Name = ' Hornby'
```

Eigenen Objekten lassen sich auch Methoden zuweisen. Dem Objekt `Autor` soll die Methode `Alter` zugewiesen werden. Diese berechnet das Alter des Autoren.

```
function Alter()
{
//Inhalt der Alter()-Funktion
}
Autor.Alter = Alter
```

Methoden werden, anders als in JavaScript üblich, ohne Klammern notiert. Das Besondere an Methoden ist nun, dass diese eine Funktion aufrufen. So könnte beispielsweise innerhalb der `Alter()`-Funktion das Alter des Autoren berechnet werden.

Um Objekte zu erzeugen, benötigen Sie einen Konstruktor sowie die **new**-Anweisung. Bei einem Konstruktor handelt es sich um eine Funktion, durch die später die Initialisierung des Objekts vorgenommen wird. Die **new**-Anweisung ist ein reserviertes Schlüsselwort, mit dem in JavaScript neue Objekte definiert werden. Die folgende Syntax beschreibt, wie das Objekt `Bibliothek` definiert wird. Zusätzlich wird hier demonstriert, wie einem Objekt mehrere Eigenschaften bzw. Methodeen zugewiesen werden können.

```
function Bibliothek(Name,Vorname,Buch)
{
this.Name = Name;
this.Vorname = Vorname;
this.Buch = Buch;
}
hornby = new Bibliothek("Hornby","Nick","Fever Pitch")
```

Im Beispiel wird mit dem **new**-Operator das Objekt `Bibliothek` definiert. Diesem werden die innerhalb der Funktion `Bibliothek()` definierten Eigenschaften `Name`, `Vorname` und `Buch` zugewiesen. Eine weitere Besonderheit an diesem Beispiel ist das Schlüsselwort **this**. Dieses gibt an, dass sich die Eigenschaften auf das aktuelle Objekt, also `Bibliothek`, beziehen. Wie sich das erzeugte Objekt verwenden lässt, zeigt die folgende Syntax.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Bibliothek(Name,Vorname,Buch)
{
this.Name = Name;
this.Vorname = Vorname;
this.Buch = Buch;
}
hornby = new Bibliothek("Hornby","Nick","Fever Pitch")
for (i in hornby)
{
document.write(i + ": ");
document.write( hornby[i] + "<BR>");
}
//-->
</SCRIPT>
```

```
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.30: Demonstration der Generierung eines eigenen Objekts

Das Beispiel schreibt die Eigenschaften sowie die dazugehörenden Werte dynamisch in das Dokument. Hierzu wird mittels einer **for**-Schleife auf das Objekt zugegriffen und werden alle Eigenschaften durchlaufen. Es ist jetzt ein Leichtes, weitere Objekte anzulegen. So könnte die Definition anderer Objekte für das gezeigte Beispiel folgendermaßen aussehen:

```
hornby = new Bibliothek("Hornby","Nick","Fever Pitch")
roth = new Bibliothek("Roth","Nick","Der menschliche Makel")
king = new Bibliothek("Hornby","King","Der letzte Kick")
```

Werden weitere Objekte erzeugt, muss diesen jeweils ein eindeutiger Name zugewiesen werden. Zudem ist darauf zu achten, dass die Anzahl der Eigenschaften mit denen der Funktionsparameter übereinstimmt.

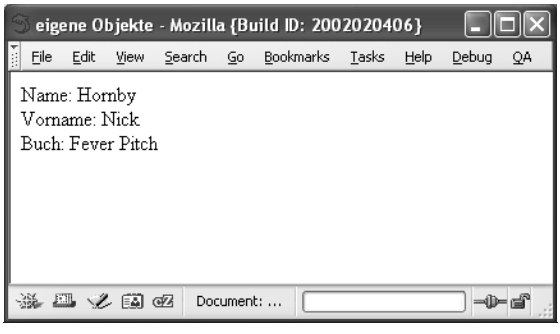


Abbildung 4.10: Die Ausgabe des hornby-Objekts

4.7.4 Objekte – Methoden und Eigenschaften

Objekte können Eigenschaften besitzen. So kann das selbst definierte Objekt *Auto* beispielsweise die Eigenschaften *Name*, *Farbe* und *Laenge* besitzen. Ebenso wie selbst definierte haben auch vordefinierte Objekte Eigenschaften. Das Objekt **images** besitzt z.B. die Eigenschaft **border**, um den Rahmen einer Grafik auszulesen oder zu verändern.

Objekt-Eigenschaften

Fehlermeldung bei falscher Zuweisung

Hinter dem Objekt wird durch Punkt getrennt die Eigenschaft notiert. Beachten Sie, dass nur solche Eigenschaften verwendet werden dürfen, die für das Objekt auch tatsächlich gültig sind. Eigenschaften, die

fälschlicherweise einem Objekt zugewiesen wurden, welches diese nicht kennt, führen zu Fehlermeldungen. Ein Beispiel, wie sich die **lastModified**-Eigenschaft des **document**-Objekts nutzen lässt, beschreibt die nachstehende Syntax. Hier wird das Datum der letzten Änderung der Datei dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<P>Seiteninhalt....</P>
<SCRIPT type="text/JavaScript">
<!--
document.write("letzte Aktualisierung: " + document.lastModified);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 4.31: Die Verwendung der lastModified-Eigenschaft des document-Objekts

Der Script-Bereich wird direkt mit dem Einlesen des Dokuments ausgeführt. Die Eigenschaft **lastModified** wird durch einen Punkt getrennt hinter den Objektnamen **document** notiert.

Objekt-Methoden

Unterschied zwischen Funktionen und Methoden

Objekte können Methoden besitzen. Dies gilt sowohl für selbst definierte wie auch für vordefinierte Objekte. Methoden sind Funktionen, die mit einer Klasse von Objekten verknüpft sind. Ebenso wie Funktionen lösen auch Methoden eine bestimmte Aktion aus. Warum aber eine Unterscheidung zwischen Methoden und Funktionen? Der Unterschied zwischen Funktionen und Methoden besteht darin, dass Methoden an ein bestimmtes Objekt gebunden sind. Die folgende allgemein gültige Syntax wird im Zusammenhang mit Methoden verwendet:

```
Objekt.Methode([Parameter],[Parameter])
```

Hinter dem Namen des Objekts wird durch einen Punkt getrennt die Methode notiert. Beachten Sie, dass jede Methode, und hier sind wir bei den Gemeinsamkeiten zwischen Funktionen und Methoden, mit einer öffnenden und schließenden Klammer versehen werden muss. Einige Methoden erwarten keine, einen oder mehrere Parameter. Es existieren aber auch solche, bei denen die Parameterzuweisung optional ist. Um einer Methode einen Parameter zuzuweisen, werden die entsprechenden Werte innerhalb der Klammer notiert. Im folgenden Beispiel wird die **select()**-Methode des **forms**-Objekts verwendet. Als Folge dieser Syn-

tax wird dem Anwender ein mehrzeiliges Eingabefeld mit einem vordefinierten Inhalt angezeigt. Die **select()**-Methode sorgt in diesem Beispiel dafür, dass der Feldinhalt durch Anklicken eines Hyperlinks markiert wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function TextMarkieren()
{
 document.formular.Feld.select();
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="formular">
<TEXTAREA rows="5" cols="30" name="Feld">
Der Verrat, Der Mann des Jahrhundert
</TEXTAREA>
</FORM>
<A href="javascript:TextMarkieren()"> ausw&auml;hlen</A>
</BODY>
</HTML>
```

*Listing 4.32: Demonstration der Verwendung der **select()**-Methode des **document**-Objekts*

Das Beispiel verdeutlicht, dass einige Methoden auch auf Unterobjekte angewandt werden können. So würde die Selektierung eines Formulars durch **select()** beispielsweise nichts bringen. Wird diese Methode aber auf ein Unterobjekt des **forms**-Objekts angewandt, tritt der gewünschte Effekt ein. Das Unterobjekt in diesem Beispiel ist `Feld`. Hierbei handelt es sich um das mehrzeilige Eingabefeld innerhalb des Formulars.

Abbildung 4.11: Der selektierte Text nach Ausführung der Funktion

4.7.5 Objekte abfragen

Ursachen für Fehlermeldungen

Durch unterschiedliche Interpretationen von Objekten und die Sprachunterschiede der verschiedenen Browser kommt es im Hinblick auf JavaScript häufig zu Fehlermeldungen. So geben WWW-Browser unter anderem immer dann eine Fehlermeldung aus, wenn ein Objekt angesprochen wird, welches der Browser nicht interpretieren kann. So schlecht diese Tatsache auch für den Entwickler sein mag, auch diese Browsereigenschaft bietet Vorteile. So lässt die Unterstützung bzw. fehlende Unterstützung eines Objekts darauf schließen, welcher Browser vom Anwender verwendet wird. Zwar ist diese Methode nicht „wasserdicht", da die Browser in verschiedenen Versionen unterschiedliche Objekte interpretieren, dennoch lässt sich eine grobe Unterscheidung der Browser durchführen. Um eine solche Unterscheidung durchzuführen, wird die **`if`**-Abfrage verwendet. Hierdurch wird es möglich, das Vorhandensein eines Objekts zu überprüfen. Ist dieses dem Browser bekannt, wird der Wert **`true`** zurückgeliefert. Anderenfalls liefert der Browser den Wert **`false`**. In Verbindung mit einer **`if-else`**-Abfrage lässt sich dieses Verhalten nutzen, um beispielsweise verschiedene Objekte für unterschiedliche Browser einzusetzen. Ebenso ist es aber auch möglich, den Anwender auf eine für seinen Browser optimierte Seite zu führen. Im folgenden Beispiel wird überprüft, ob der verwendete Browser das **`all`**-Objekt kennt. Je nach Rückgabewert findet eine automatische Weiterleitung auf die Seite `explorer.html` bzw. `netscape.html` statt.

```
<HTML>
<HEAD>
</HEAD>
</BODY>
<SCRIPT type="text/JavaScript">
<!--
if(document.all)
 window.location.href = "explorer.html";
else
 window.location.href = "netscape.html";
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 4.33: Eine simple Browserweiche

Das gezeigte Script wird direkt mit dem Einlesen der Datei ausgeführt. Als **`if`**-Bedingung wird innerhalb der Klammern das zu suchende Objekt, in diesem Beispiel **`all`**, notiert. Wird bei dieser **`if`**-Abfrage der Wert **`true`** zurückgeliefert, kann der Browser dieses Objekt also interpretieren, wird auf die Seite `explorer.html` weitergeleitet. Für den Fall, dass das Ob-

jekt nicht interpretiert werden kann, erfolgt eine automatische Weiterleitung auf die Seite `netscape.html`.

Voraussetzung für Browserweichen

Voraussetzung für diese Form der Browsererkennung ist selbstverständlich die Kenntnis darüber, welcher Browser in welcher Version das entsprechende Objekt interpretieren kann. Im vorgestellten Beispiel wurde für die Browserunterscheidung das **all**-Objekt verwendet. Da dieses Objekt ausschließlich vom Internet Explorer unterstützt wird, kann eine Browserunterscheidung hierauf beruhen. Selbstverständlich ist es so, dass wohl nur den wenigsten Programmierern alle Objekte und deren Verfügbarkeit in den verschiedensten Browsern geläufig sind. Aus diesem Grund kann das zuvor gezeigte Script durchaus zu Testzwecken verwendet werden.

4.7.6 Mehrere Anweisungen

speziellere Syntax

Im Zusammenhang mit Objekten kann es vorkommen, dass Sie mehrere Anweisungen notieren, die alle mit dem gleichen Objekt arbeiten. Selbstverständlich können Sie für diesen Zweck die bisher gezeigten Syntaxformen verwenden. Um effizienteres Arbeiten zu ermöglichen, existiert hierfür jedoch auch eine spezielle Syntax.

```
with(Objekt)
{
Eigenschaft
Eigenschaft
Methode()
}
```

Hinter dem einleitenden Schlüsselwort **with** wird der Objekt-Name notiert. Bei diesem Objekt kann es sich beispielsweise um das **document**-Objekt handeln. Es ist aber ebenso gestattet, Unterobjekte auf diese Weise anzusprechen. Innerhalb der geschweiften Klammern werden alle erforderlichen Eigenschaften und Methoden des Objekts notiert. Zur Veranschaulichung dient die nachstehende Syntax. Hierin wird in zwei nacheinander auftauchenden Meldungsfenstern zuerst die Breite und anschließend die Höhe der eingebundenen Grafik angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeGrafik()
{
with(document.images[0])
{
```

```
alert(width)
alert(height)
}
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<IMG src="grafix.gif" width="300" height="200" name="Grafik">
<BR>
<A href="javascript:zeigeGrafik()">Eigenschaften und Methoden</A>
</BODY>
</HTML>
```

Listing 4.34: Die beiden `alert()`-Methoden geben die Breite und Höhe der ersten Grafik aus.

Die Funktion `zeigeGrafik()` wird durch Anklicken des Hyperlinks ausgelöst. Hinter dem **`with`**-Schlüsselwort wird das Objekt **`document.images[0]`** notiert. Bei diesem Objekt handelt es sich um die erste innerhalb des Dateikörpers vorkommende Grafik. Innerhalb der geschweiften Klammern werden zwei **`alert()`**-Methoden notiert. Diese erhalten als Werte die beiden Eigenschaften **`width`** und **`height`**. Anhand dieser Syntax wird deutlich, dass diese Schreibweise durchaus ihre Reize hat. Zum einen wird der Quellcode übersichtlicher und zum anderen kommen Sie mit weniger JavaScript-Code aus. Tatsächliche Auswirkungen zeigt diese Notationsform jedoch erst im Zusammenhang mit umfangreichen JavaScript-Programmen.

4.7.7 Auf ein Objekt Bezug nehmen

verkürzte Syntax

In einigen Fällen steht für den Browser fest, auf welches Objekt sich eine bestimmte JavaScript-Anweisung bezieht. Ist dem Programmierer diese Tatsache bekannt, kann dies für eine verkürzte Schreibweise und somit für eine weniger komplexe Syntax genutzt werden. Um eine solch minimierte Schreibweise umsetzen und einsetzen zu können, steht innerhalb von JavaScript das Schlüsselwort **`this`** bereit. Durch dessen Einsatz wird dem JavaScript-Programm mitgeteilt, dass sich der Befehl auf das aktuelle Objekt bezieht. Ein Beispiel hierzu:

```
<FORM name="stand">
<INPUT type="text" name="plausi">
<INPUT type="button" value="prüfen" onclick="alert(this.form.plausi.value)">
</FORM>
```

Innerhalb des gezeigten Beispiels werden ein Texteingabefeld und ein Button anhand herkömmlicher HTML-Syntax definiert. Beide Elemente wer-

den innerhalb eines mit `stand` gekennzeichneten Formulars notiert. Die aufgeführte Syntax hat die Ausgabe der Inhalte, welche vom Nutzer innerhalb des Textfeldes eingegeben wurden, nach dem Anklicken des Buttons zur Folge. Ausgelöst wird der JavaScript-Code durch den Event-Handler **`onclick.`** Die Nutzereingaben werden hierauf in einem Meldungsfenster ausgegeben. Das gleiche Ergebnis, jedoch mit einer komplexeren Syntax, würde auch durch die folgende Anweisung erzielt werden können.

```
onclick="alert(document.stand.plausi.value)"
```

Hier wird explizit der Name des Formulars, nämlich `stand`, aufgeführt. Wie dieses Beispiel zeigt, ist die erste der gezeigten Lösungen, die sich des **`this`**-Schlüsselwortes bedient, nicht nur effizienter, sondern trägt zudem zu einer erhöhten Übersichtlichkeit der Syntax bei. Ermöglicht wird der Einsatz von **`this`** in diesem Beispiel, da sich der entsprechende JavaScript-Aufruf innerhalb des aktuellen Formulars befindet und demzufolge auf dieses direkt Bezug genommen werden kann.

4.7.8 Objekte löschen

Speicherplatz freigeben

Einmal erzeugte Objekte werden häufig nicht während des gesamten Programmablaufs benötigt. In solchen Fällen wird wertvoller Speicherplatz belegt. Durch den Einsatz des Schlüsselwortes **`null`** kann der von einem Objekt belegte Speicherplatz wieder freigegeben werden.

```
var meinObjekt = new function();
…
meinObjekt = null;
```

Im ersten Teil des Script-Fragments wird das Objekt `meinObjekt` erzeugt. Um den von diesem Objekt belegten Speicherplatz wieder freigeben zu können, wird diesem Objekt der Wert **`null`** zugewiesen.

delete-Operator verwenden

Mittels des **`delete`**-Operators kann ein Objekt, eine Objekt-Eigenschaft oder ein Element innerhalb eines Arrays gelöscht werden. Sinnvoll ist dies vor allem vor dem Hintergrund, dass Objekte nicht während des gesamten Programms benötigt werden. Die Funktionalität des **`delete`**-Operators soll anhand eines Beispiels veranschaulicht werden.

```
Wert1=40
var Wert2= 56
eigenes=new Number()
eigenes.Wert3=4
```

Die nachstehende Tabelle zeigt verschiedene Anwendungen des **delete**-Operators. Die linke Spalte beinhaltet mögliche Anweisungen, mit denen verschiedene Elemente der aufgeführten Syntax gelöscht werden sollen. Die rechte Spalte beschreibt, ob dies gelingen kann.

Anweisung	Ergebnis
delete Wert1	Kann gelöscht werden.
delete Wert2	Kann nicht gelöscht werden, da das Objekt mit **var** deklariert wurde.
delete Math.PI	Vordefinierte Eigenschaften können nicht gelöscht werden.
delete eigenes.Wert3	Eigene Eigenschaften können gelöscht werden.
delete eigenes	Objekt kann gelöscht werden.

*Tabelle 4.14: Auswirkungen des **delete**-Operators*

Arrays löschen

Sie können den **delete**-Operator auch zum Löschen von Array-Elementen einsetzen. Hierbei ist zu beachten, dass die Array-Länge hiervon unberührt bleibt. Nehmen wir an, dass Sie innerhalb eines Arrays das dritte Element löschen. In diesem Fall ist das dritte Element tatsächlich gelöscht, das vierte ist jedoch nach wie vor das vierte Element.

4.8 Arrays

Typgleiche Variablen werden in Feldern gespeichert.

Die bislang vorgestellten Datentypen wie beispielsweise Strings und boolesche Werte reichen für die Lösung komplexer Aufgaben häufig nicht aus. Und selbst wenn sich durch deren Verwendung eine Lösung finden lässt, ist diese meist zu umständlich und zeugt von schlechtem Programmierstil. In solchen Fällen empfiehlt sich die Verwendung von Arrays. Bei Arrays handelt es sich um Felder, in denen typgleiche Variablen gespeichert werden können. JavaScript stellt für das Erzeugen von Feldern das **Array()**-Objekt bereit. Arrays können auf die folgenden Arten erzeugt werden. Beachten Sie, dass wir im Laufe dieses Abschnitts auf die unterschiedlichen Array-Varianten noch genauer eingehen werden.

```
Arrayname = new Array()
```

Eine Array-Objektinstanz wird in einem Objekt mit einem frei wählbaren Namen gespeichert. Hinter dem Gleichheitszeichen folgen das Schlüsselwort **new** und das Objekt **Array()**. Diese Syntax wird verwendet, wenn Sie während der Programmierung noch nicht die Anzahl der im Array enthaltenen Elemente wissen.

```
Arrayname = new Array (Zahl)
```

Wenn Sie wissen, wie viele Elemente in einem Array gespeichert werden sollen, verwenden Sie diese Syntax. Auch hier wird die Array-Objektinstanz in einem Objekt mit einem frei wählbaren Namen gespeichert. Hinter dem Gleichheitszeichen folgen das Schlüsselwort **new** und das **Array()**-Objekt. Als Parameter wird hier eine Zahl übergeben. Die Anweisung **Array(**4**)** hat die Erzeugung eines Arrays mit vier Elementen zur Folge.

```
Arrayname = new Array(Element0, Element1, Element_n)
```

Wenn Sie die Anzahl der Elemente sowie deren Wert kennen, können Sie diese Syntax verwenden. Dem **Array()**-Objekt werden die einzelnen Elemente als Parameter übergeben.

Nachdem Sie nun wissen, wie sich Arrays erzeugen lassen, wenden wir uns dem Zugriff auf diese zu. Auch hierfür existieren wieder verschiedene Varianten. Das folgende Beispiel demonstriert, wie Sie das i-te Element eines Arrays ansprechen und dessen Wert in einem Meldungsfenster ausgeben können.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
Werte = new Array(100,110,120,200,300);
var Ausgabe = Werte[3];
alert(Ausgabe);
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<A href="javascript:zeigeWert()">zeige Eintrag</A>
</BODY>
</HTML>
```

Listing 4.35: Ausgabe der vierten Zahl des Arrays

Die Instanz des **Array()**-Objekts wird in dem Namen `Werte` gespeichert. Als Parameter werden dem **Array()**-Objekt *5* Zahlen übergeben. Der Variablen `Ausgabe` wird **Werte[**3**]** zugewiesen. Hierdurch wird die vierte Zahl des `Werte`-Arrays gespeichert. Beachten Sie, dass die interne Zählung bei `0` beginnt. Möchten Sie also beispielsweise den ersten Wert ansprechen, müssen Sie `Werte[0]` notieren. Abschließend wird der Wert der Variablen `Ausgabe`, also `200`, über die **alert()**-Methode ausgegeben.

4.8.1 Mehrdimensionale Arrays

beliebig viele Dimensionen

Arrays können ein- und mehrdimensional definiert werden. Ein eindimensionaler Array würde beispielsweise über **Array(**200**)** erzeugt werden. Eindimensionale Arrays genügen aber häufig nicht den Anforderungen, die an ein JavaScript-Programm gestellt werden. Oftmals besser geeignet sind mehrdimensionale Arrays. Ein solcher Array kann z.B. über **Array(**200,300**)** erzeugt werden. Diese Syntax bewirkt, dass ein zweidimensionaler Array erstellt wird. Sie können einem Array beliebig viele Dimensionen zuweisen. Das folgende Beispiel nutzt einen mehrdimensionalen Array zur Ausgabe von Text in Tabellenform.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Tabelle = new Array(5)
for (var i=0; i < Tabelle.length; ++i)
 Tabelle[i] = new Array(4);
Tabelle[3][0] = "<table border=1>";
Tabelle[3][1] = "<TR><TD>";
Tabelle[3][2] = "Herzlich Willkommen";
Tabelle[3][3] = "</TD></TR>";
Tabelle[3][4] = "</TABLE>";
document.write(Tabelle[3][0]);
document.write(Tabelle[3][1]);
document.write(Tabelle[3][2]);
document.write(Tabelle[3][3]);
document.write(Tabelle[3][4]);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 4.36: Die Generierung einer Tabelle mittels eines Arrays

Im Beispiel wird in dem Objektnamen `Tabelle` ein neuer Array mit fünf Elementen gespeichert. Die **for**-Schleife erstellt für jedes der fünf Elemente einen neuen Array. Diese Arrays sind zunächst leer. Im Beispiel soll der fünfte Array mit Inhalten gefüllt werden. Hierzu wird dieser jeweils über `Tabelle[3]` angesprochen. Um die einzelnen Elemente zu füllen, werden diese ebenfalls über einen numerischen Wert angesprochen. Soll also das zweite Element mit Inhalten gefüllt werden, muss die Anweisung `[1]` notiert werden. Als Wertzuweisungen werden in dem Beispiel die Tags für die Erzeugung einer Tabelle sowie der anzuzeigende

Text definiert. Abschließend werden die Elemente des vierten Arrays über die **write()**-Methode ausgegeben.

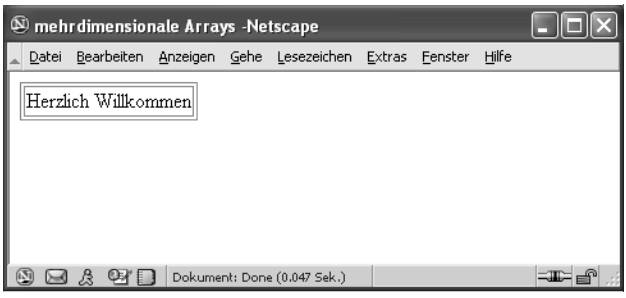


Abbildung 4.12: Die Ausgabe der Tabelle

4.8.2 Assoziative Arrays

Zugriff durch eine Zeichenkette

Bislang haben wir auf das i-te Element eines Arrays über einen numerischen Wert zugegriffen. Durch die Verwendung von assoziativen Arrays kann der Zugriff aber auch mittels einer Zeichenkette erfolgen. Sie werden assoziative Arrays im Lauf dieses Buches noch zur Genüge kennen lernen. Schließlich bedienen sich einige JavaScript-Objekte dieser Möglichkeit. Grundsätzlich sind alle JavaScript-Objekte, welche die Eigenschaft **id** oder **name** besitzen, assoziative Arrays. Der Vorteil dieser Arrays ist zweifellos, dass Sie als Programmierer nicht die Position eines Elements kennen müssen, um dessen Wert zu ermitteln. Dies geschieht über den Element-Namen. Im folgenden Beispiel wird das Array `Band` erzeugt. Dieses besteht aus zwei Elementen. Beide Elemente stellen jeweils selbst ein Array dar. Um auf die Elemente dieser Arrays zugreifen zu können, werden Namen vergeben. Als Ergebnis dieser Syntax werden die Werte beider Arrays in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Band=new Array();
 Band[0] = new Array();
 Band[0]["Name"] = "The Offspring";
 Band[0]["LP"] = "Ixnay";
 Band[0]["LP2"]= "Live";
 Band[1]=new Array()
 Band[1]["Name"] = "Green Day";
 Band[1]["LP"] = "International Superhits";
 Band[1]["LP2"]= "Dookie";
 for (var i=0;i<Band.length;i++)
 {
```

```
   for (var Beschreibung in Band[i])
      document.write(Beschreibung + ": "+Band[i][Beschreibung]+"<BR>");
  document.write("<BR>");
 }
//-->
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.37: Bandnamen und LPs werden ausgegeben.

Über die **for**-Schleife werden die Elemente des Arrays und deren Eigenschaften durchlaufen. Als Abbruchkriterium dient die Anzahl der vorhandenen Arrays. Im nächsten Schritt wird eine **for-in**-Schleife eingesetzt. Diese greift auf alle Eigenschaften eines Arrays zu und schreibt diese dynamisch in das Dokument. Sind alle Eigenschaften erfasst, beginnt der Schleifendurchlauf erneut. Nach dem zweiten Durchlauf wird die Schleife abgebrochen.

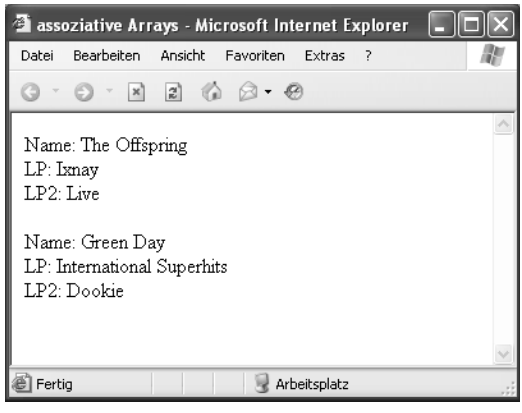


Abbildung 4.13: Ausgabe des Array-Inhalts

4.9 Ereignisbehandlung – Event-Handling

auf Ereignisse reagieren

Die Ereignisbehandlung stellt eines der elementarsten Bindeglieder zwischen JavaScript und HTML dar. Hierdurch wird es möglich, mit dem Eintreten eines bestimmten Ereignisses (Event) einen definierten JavaScript-Code auszuführen. Das hierbei einzusetzende Werkzeug lautet Event-Handler. Hierbei handelt es sich um fest vorgeschriebene Schlüsselwörter, die jeweils ein mögliches Ereignis beschreiben. So handelt es sich beim Anklicken eines Hyperlinks beispielsweise um ein Ereignis. Der dieses Ereignis beschreibende Event-Handler lautet **onclick**. Neben

diesem Event-Handler existieren zahlreiche andere, die im folgenden Abschnitt vorgestellt werden. Zunächst jedoch eine Syntax, welche die Anwendung von Event-Handlern beschreibt.

```
<IMG src="follow.jpg" height="200" width="300" alt="folge mir"
onmouseover="alert('Sie wollen die Grafik also nicht sehen.....')">
```

In dem gezeigten Beispiel wird eine Grafik eingebunden. Fährt der Anwender über diese mit der Maus, wird ein Text in einem Meldungsfenster angezeigt. Der in diesem Beispiel verwendete Event-Handler lautet **onmouseover**. Dieser wird als Attribut innerhalb des **<IMG>**-Tags notiert. Als Wert bekommt der Event-Handler die **alert()**-Methode zugewiesen. Diese dient dazu, ein Meldungsfenster zu öffnen. Als Wert wird dieser Methode der innerhalb des Meldungsfensters anzuzeigende Text übergeben.

4.9.1 Event-Handler im Überblick

Es gibt noch andere Event-Handler.

Die folgende Tabelle zeigt ausschließlich die Event-Handler, die im HTML-Standard erwähnt sind. Beachten Sie, dass noch andere existieren. Bei diesen handelt es sich jedoch zumeist um browserspezifische Event-Handler, die häufig nur von einem Browser unterstützt werden. Nähere Informationen zu solchen Event-Handlern finden Sie auf den Entwicklerseiten des jeweiligen Browserherstellers.

Event-Handler	Beschreibung
onabort	Code wird beim Abbruch des Ladens einer Seite ausgeführt.
onblur	Code wird dann ausgeführt, wenn ein Element den Fokus verliert.
onchange	Code wird ausgeführt, wenn ein Element einen veränderten Wert erhält.
onclick	Code wird ausgeführt, wenn ein Element angeklickt wird.
ondblclick	Beim doppelten Anklicken eines Elements wird der Code ausgeführt.
ondragdrop	Wird ein Objekt über das Browserfenster gezogen und dort losgelassen, löst dies den Code aus.
onerror	Löst ein Element eine Fehlermeldung aus, wird der Code ausgeführt.
onfocus	Löst den Code aus, wenn ein Element aktiviert wird.
onkeydown	Bei Tastendruck wird der Code ausgeführt.
onkeypress	Code wird ausgeführt, wenn eine Taste gedrückt und gedrückt gehalten wird.
onkeyup	Code wird ausgeführt, wenn eine Taste gedrückt und wieder losgelassen wird.

Tabelle 4.15: Event-Handler

Event-Handler	Beschreibung
onload	Beim Laden einer Datei wird der Code ausgelöst.
onmousedown	Code wird ausgeführt, wenn die Maustaste gedrückt wird.
onmousemove	Beim Bewegen der Maus wird der Code ausgeführt.
onmouseout	Wird mit der Maus über ein Element gefahren und dieses wieder verlassen, wird der Code ausgeführt.
onmouseover	Code wird beim Überfahren eines Elements mit der Maus ausgelöst.
onmouseup	Wird die Maustaste gedrückt und wieder losgelassen, löst dies den Code aus.
onmove	Beim Bewegen eines Elements wird der Code ausgelöst.
onreset	Code wird ausgeführt, wenn Formularinhalte gelöscht werden sollen.
onresize	Wird die Größe des Browserfensters verändert, löst dies den Code aus.
onselect	Beim Selektieren von Text wird der Code ausgeführt.
onsubmit	Der Code wird beim Absenden eines Formulars ausgelöst.
onunload	Beim Verlassen einer Seite wird der Code ausgelöst.

Tabelle 4.15: Event-Handler (Forts.)

Beachten Sie, dass besonders der Netscape Navigator 4 Event-Handler nicht so handhabt, wie dies vom W3C vorgesehen wurde. Der Internet Explorer ist hingegen im Umgang mit Event-Handlern sehr nah am W3C-Standard.

4.9.2 Ereignisse im Internet Explorer

DOM sehr ausgereift

Das DOM (Document Objekt Model) des Internet Explorers ist bereits jetzt sehr ausgereift. Es lassen sich hier beispielsweise alle Tags mit Event-Handlern versehen. Beachten Sie, dass diese Vorgehensweise im Netscape Navigator 4 nicht funktioniert. Um Seiten für die beiden „großen" Browser zu entwickeln, sollten also nur solche Tags mit Event-Handlern versehen werden, bei denen dies auch vom Netscape Navigator interpretiert werden kann.

Ereignisse als Objekteigenschaften

veränderte Syntax

Das Ereignismodell von JavaScript hat sich grundlegend verändert. So müssen beispielsweise keine HTML-Event-Handler mehr eingesetzt werden, um auf ein bestimmtes Ereignis reagieren zu können. Es soll an dieser Stelle keine ausführliche Erörterung der veralteten, aber dennoch auch heute noch einsetzbaren Variante der Ereignisbehandlung geben. Lediglich das Prinzip und die hieraus resultierenden Neuerungen soll-

ten verstanden werden. Ein JavaScript-Programm kann auf herkömmliche Art definiert und anschließend durch das Setzen eines HTML-Event-Handlers ausgeführt werden. Ein simples Beispiel soll diese Vorgehensweise zeigen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Buchtitel()
{
  alert("Der Mannn des Jahrhunderts")
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="Buchtitel()">
</BODY>
</HTML>
```

Listing 4.38: Event-Handling auf herkömmliche Weise

Das Prinzip ist denkbar einfach: Der Event-Handler **onload** wird hier als HTML-Attribut des **<BODY>**-Tags spezifiziert. Als Wert wird diesem Attribut der Funktionsaufruf zugewiesen. Tritt der Event-Handler in Kraft, wird die Funktion `Buchtitel()` ausgeführt. Um an dieser Stelle keine Missverständnisse aufkommen zu lassen: Das zuvor gezeigte Beispiel ist auch nach wie vor einsetzbar und in seiner Syntax völlig korrekt. Dennoch existiert seit der JavaScript-Version 1.1 eine Notationsmöglichkeit, mit deren Hilfe sich Event-Handler ohne HTML-Code einsetzen lassen. Wie sich dies darstellt, soll anhand des folgenden Beispiels veranschaulicht werden. Im Ergebnis stellen sich beide zwar gleich dar, so wird auch durch die nachstehende Syntax ein Meldungsfenster mit einer entsprechenden Nachricht ausgegeben, die Herangehensweise unterscheidet sich dennoch grundlegend.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Buchtitel()
{
alert("Der Mannn des Jahrhunderts")
}
window.onload = Buchtitel
//-->
</SCRIPT>
```

```
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.39: Die moderne Variante des Event-Handlings

Auf zwei Aspekte dieser Notationsvariante muss aufmerksam gemacht werden. Die Event-Handler müssen klein geschrieben werden, und die Zuweisung einer Funktion darf sich nicht der beiden angehängten Klammern bedienen. Denn würden die Klammern gesetzt, führte dies zum sofortigen Auslösen der Funktion. In diesem Fall soll die Funktion jedoch erst dann ausgeführt werden, wenn das Ereignis **`onload`** eintritt. Und eben dies wird mit der gezeigten Syntax erreicht.

Ereignisse mit event

Spezielles im Internet Explorer

Eine recht interessante Methode bezüglich der Ereignisbehandlung hält der Internet Explorer bereit. Problematisch ist die hier gezeigte Syntaxform aber dennoch. Sie kann lediglich vom Internet Explorer interpretiert werden. Für Crossbrowser-Lösungen scheidet diese Notationsvariante demzufolge aus. Und dennoch: Dank der simplen Syntax und der Logik, die sich hinter dieser Entwicklung verbirgt, soll diese besondere Form der Ereignisbehandlung kurz vorgestellt werden. Im folgenden Beispiel wird ein herkömmlicher HTML-Button definiert. Klickt ein Anwender auf diese Schaltfläche und wird somit das Ereignis **`onclick`** ausgelöst, gibt der Internet Explorer ein Meldungsfenster mit einem Hinweistext aus.

```
<HTML>
<HTML>
<SCRIPT type="text/JavaScript"
event="onclick" for="public">
<!--
alert ("Spezielles im Internet Explorer");
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM>
<INPUT type="button" value="anklicken" name="public">
</FORM>
</BODY>
</HTML>
```

*Listing 4.40: Verwendung der beiden Attribute **`event`** und **`for`***

Das Besondere dieser Form der Ereignisbehandlung ist die zunächst augenscheinlich auffallende Notationsvariante. Innerhalb des **<SCRIPT>**-Tags werden zusätzlich zu **type** die beiden Attribute **event** und **for** notiert. Wobei dem Attribut **event** als Wert ein im Internet Explorer anwendbarer Event-Handler zugewiesen werden muss. Das Attribut **for** erwartet einen frei wählbaren Namen, welcher in exakt der gleichen Schreibweise angegeben werden muss, wie dies auch in dem entsprechenden HTML-Tag vorgenommen wurde. Im letzten Beispiel wurde dem HTML-Button, nach dessen Anklicken das Meldungsfenster geöffnet werden soll, über das Attribut **name** der dateiweit eindeutige Bezeichner **public** zugewiesen. Und eben auf exakt diesen bezieht sich der Wert des **for**-Attributs. Das durch den Event-Handler auszulösende JavaScript-Programm wird wie gewöhnlich notiert und unterliegt keinen gesonderten Notationsvorschriften.

Abfangen von Ereignissen

keine spezielle Methode vorhanden

Um Ereignisse im Internet Explorer abfangen zu können, existiert, anders als im Netscape Navigator, keine eigene Methode. Der Internet Explorer stellt zunächst fest, ob bei einem Objekt ein Ereignis eintritt und ob für dieses ein Event-Handler notiert wurde. Ist das nicht der Fall, wird überprüft, ob ein übergeordnetes Objekt einen Event-Handler besitzt. Findet der Internet Explorer hier einen Event-Handler, wird dieser auf das Objekt angewandt. Ist in keinem der übergeordneten Objekte ein Event-Handler vorhanden, kann das Ereignis nicht abgefangen werden. Im folgenden Beispiel werden alle Mausklicks abgefangen. Dies geschieht unabhängig davon, ob diese auf dem Formular-Button oder einen anderen Bereich des Dokuments ausgeführt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Ausgabe()
{
   window.status += "klick "
}
document.onclick = Ausgabe
//-->
</SCRIPT>
</HEAD>
</BODY>
<I>Ereignisse</I>
<FORM>
<INPUT type="BUTTON" value="Klick mich!">
```

```
</FORM>
<B>im Internet Explorer</B>
</BODY>
</HTML>
```

Listing 4.41: Das Wort `klick` wird in der Statuszeile angezeigt.

Innerhalb der `Ausgabe()`-Funktion wird festgelegt, dass in der Statuszeile der Text `klick` angezeigt werden soll. Über **`document.onclick =`** `Ausgabe` wird festgelegt, dass beim Anklicken eines Dokumentbereichs die Funktion `Ausgabe()` ausgeführt wird. Problematischer stellt sich dies jedoch beim Formular-Button dar. Diesem wird kein Event-Handler zugewiesen. Hieraus würde eigentlich resultieren, dass die Funktion dort nicht ausgeführt wird. Der Internet Explorer überprüft jedoch, ob einem höhergeordneten Element ein Event-Handler zugewiesen wurde. In diesem Beispiel heißt dies, dass dem Formular-Button automatisch der Event-Handler **`onclick`** des **`document`**-Objekts zugewiesen wird.

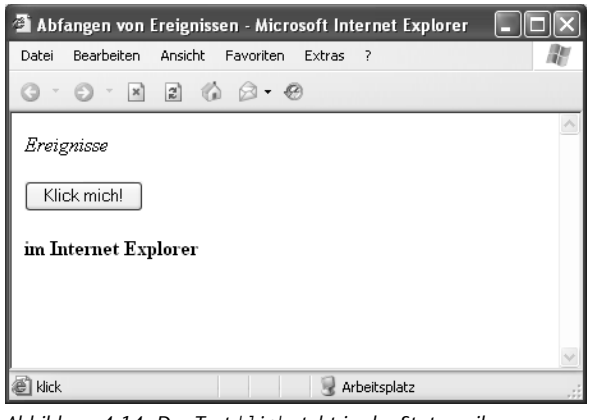


Abbildung 4.14: Der Text `klick` steht in der Statuszeile.

4.9.3 Ereignisse im Netscape Navigator

Browser-spezifisches

Wieder einmal ist es nicht möglich, eine Lösung der Ereignisbehandlung für beide Browser zu programmieren. Auch im Hinblick auf die Ereignisbehandlung bleibt wieder nur die Verwendung beider browserspezifischen Syntaxformen. So ärgerlich dies auch sein mag, nur so können Ereignisse tatsächlich in beiden Browsern verwendet werden. Nachfolgend finden Sie die Grundlagen der Ereignisbehandlung für den Netscape Navigator.

Ereignisse als Objekteigenschaften

In alten Navigator-Versionen wurden Ereignisse zumeist mit HTML-Code besetzt. Im einleitenden HTML-Tag wurden der Event-Handler und der Funktionsname notiert. Seit JavaScript 1.1 steht eine andere Variante zur Verfügung. Nachfolgend werden die beiden Syntaxformen gezeigt.

```
<BODY onload="Ladezustand()"> // herkömmliche Syntax
window.onload = Ladezustand // Syntax nach JavaScript 1.1
```

Bei der zweiten Variante fällt auf, dass dem Funktionsaufruf `Ladezustand` keine Klammern zugewiesen wurden. Dies liegt daran, dass bei dieser Syntaxform keine Funktionsaufrufe gestattet sind. Bei dieser Syntax handelt es sich vielmehr um eine Referenz auf die Funktion. Auf Grund einer solchen Notation ist man bei der Entwicklung von Programmen jetzt freier. Eine solche Funktionsreferenz kann nun an jeder Stelle des Dokuments notiert werden. Zudem kann der Event-Handler durch Zuweisung des Schlüsselworts **`null`** gelöscht werden. Im folgenden Beispiel werden die beiden Ereignisse **`onclick`** und **`onmousedown`** überwacht. Beim Eintreten dieser Ereignisse wird `geklickt` und `gedrückt` in der Statuszeile angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function klicken(){window.status += "geklickt "}
function druecken(){window.status += "gedrückt "}
function initialiseren(){
   window.status = ''
   with (document.Formular.anklickbar)
      onclick = klicken
      onmousedown = druecken
}
window.onload = init
//-->
</SCRIPT>
</HEAD>
</BODY>
<FORM name="Formular">
<INPUT type="button" name="anklickbar" value="Klick mich!">
</FORM>
</BODY>
</HTML>
```

Listing 4.42: Hier werden zwei Ereignisse überwacht.

Die beiden Funktion `klicken()` und `druecken()` sorgen dafür, dass die Texte `geklickt` bzw. `gedrückt` in der Statuszeile angezeigt werden. Die Funktion `initialisieren()` überprüft, welches der Ereignisse **onclick** und **onmousedown** auf den Formular-Button `anklickbar` angewandt wurde, und führt die entsprechende Funktion aus. Dass die Funktion `initialisieren()` erst mit dem Laden der Seite ausgeführt wird, dafür sorgt der Event-Handler **onload**.

Ereignisse überwachen

Es soll an dieser Stelle nicht nochmals ausführlich auf die Möglichkeit der Ereignisbehandlung eingegangen werden. Weiterführende Informationen hierzu erhalten Sie ab Seite 134. Hier soll lediglich die Möglichkeit des Einsatzes der Ereignisbehandlung für das **window**-Objekt vorgestellt werden. Innerhalb des angegebenen Fensters lassen sich die folgenden Ereignisse überwachen:

CaptureEvent	Event-Handler	Beschreibung
Event.ABORT	**onabort**	Das Ereignis **Event.ABORT** tritt dann ein, wenn der Anwender den Ladevorgang einer Grafik abbricht.
Event.BLUR	**onblur**	Das Ereignis **Event.BLUR** tritt dann ein, wenn ein zuvor aktiviertes Element wieder verlassen wird.
Event.CHANGE	**onchange**	Das Ereignis **Event.CHANGE** tritt ein, wenn der Wert eines Elements geändert wurde.
Event.CLICK	**onclick**	Das Ereignis **Event.CLICK** tritt dann ein, wenn der Anwender ein Element durch Anklicken betätigt.
Event.DBLCLICK	**ondblclick**	Das Ereignis **Event.DBLCLICK** tritt ein, wenn der Anwender ein Element durch doppeltes Anklicken auslöst.
Event.DRAGDROP	**ondragdrop**	Das Ereignis **Event.DRAGDROP** tritt ein, wenn ein Systemelement, wie z.B eine Datei, unter Zuhilfenahme der System-Funktion innerhalb eines Fensters platziert wird.
Event.ERROR	**onerror**	Das Ereignis **Event.ERROR** tritt dann ein, wenn eine Grafik vom Browser nicht geladen werden kann.
Event.FOCUS	**onfocus**	Das **Event.FOCUS**-Ereignis tritt dann ein, wenn der Anwender ein Element aktiviert.
Event.KEYDOWN	**onkeydown**	Das **Event.KEYDOWN**-Ereignis tritt dann ein, wenn der Anwender während der Aktivierung eines Elements eine Taste drückt.

*Tabelle 4.16: Ereignisse für das **window**-Objekt*

CaptureEvent	Event-Handler	Beschreibung
Event.KEYPRESS	onKeypress	Das Ereignis **Event.KEYPRESS** tritt ein, wenn der Anwender innerhalb eines zuvor aktivierten Elements eine Taste drückt.
Event.KEYUP	onkeyup	Das Ereignis **Event.KEYUP** tritt ein, wenn der Anwender innerhalb eines aktivierten Elements eine Taste drückt und diese anschließend wieder los lässt.
Event.LOAD	onload	Das Ereignis **Event.LOAD** tritt ein, wenn eine Datei geladen wird.
Event.MOUSEDOWN	onmousedown	Das Ereignis **Event.MOUSEDOWN** tritt ein, wenn der Anwender die Maustaste nach dem Anklicken gedrückt hält.
Event.MOUSEMOVE	onmousemove	Das Ereignis **Event.MOUSEMOVE** tritt ein, wenn der Anwender die Maus über einem bestimmten HTML-Element bewegt.
Event.MOUSEOUT	onmouseout	Das Ereignis **Event.MOUSEOUT** tritt ein, wenn der Anwender über ein Element fährt, dieses jedoch wieder verlässt.
Event. MOUSEOVER	onmouseover	Das Ereignis **Event.MOUSEOVER** tritt ein, wenn der Anwender mit der Maus über ein Element der HTML-Seite fährt.
Event.MOUSEUP	onmouseup	Das Ereignis **Event.MOUSEUP** tritt ein, wenn der Anwender ein Element durch Anklicken markiert und die Maustaste anschließend wieder loslässt.
Event.MOVE	onmove	Das Ereignis **Event.ONMOVE** tritt ein, wenn der Anwender ein Element bewegt.
Event.RESET	onreset	Das Ereignis **Event.RESET** tritt ein, wenn der Anwender die Eingaben innerhalb eines Formulars zu löschen versucht.
Event.RESIZE	onresize	Das Ereignis **Event.RESIZE** tritt ein, wenn die Größe des Browserfensters verändert wird.
Event.SELECT	onselect	Das Ereignis **Event.SELECT** tritt ein, wenn der Anwender ein Wort oder eine Textpassage markiert.
Event.SUBMIT	onsubmit	Das Ereignis **Event.SUBMIT** tritt ein, wenn der Anwender die Daten eines Formulars abzusenden versucht.
Event.UNLOAD	onunload	Das Ereignis **Event.UNLOAD** tritt ein, wenn der Anwender die aktuell besuchte Internetseite verlässt.

Tabelle 4.16: Ereignisse für das window-Objekt (Forts.)

Die Schreibweise der Anwenderereignisse weicht von der Notation von Event-Handlern leicht ab. Die Unterschiede betreffen das Nichtnotieren des **on** vor jedem Ereignis und das stattdessen zu notierende **Event-**

Objekt. Als Beispiel sei hierfür der bekannte **onclick**-Event-Handler genannt. Dessen Syntax müsste im Zusammenhang mit Anwenderereignissen in **Event.CLICK** umgewandelt werden. Das folgende Beispiel beschreibt eine typische Syntax, die so im Zusammenspiel mit **captureEvents()** häufig verwendet wird – das „Deaktivieren" der rechten Maustaste.

```
<HTML>
<HEAD>
<SCRIPT type="text/javascript">
<!--
function sperren()
{
function rechts(e)
{
if (window.Event)
return false;
else
{
alert("Kein Rechtsklick möglich");
return false;
}
return true;
}
document.onmousedown=rechts;
if (document.layers)
window.captureEvents(Event.MOUSEDOWN);
window.onmousedown=rechts;
}
// -->
</SCRIPT>
</HEAD>
<BODY onload="sperren()">
</BODY>
</HTML>
```

Listing 4.43: Keine Anzeige des Quelltextes durch die Verwendung der Maustaste möglich

In dem gezeigten Beispiel wird das Drücken der rechten Maustaste überwacht. Gesetzt den Fall, dieses Ereignis tritt ein, wird innerhalb eines Meldungsfensters ein Warnhinweis ausgegeben. Beachten Sie, dass die gezeigte Syntax lediglich im zweiten Teil auf das in diesem Abschnitt gezeigte **captureEvents()** eingeht, wie dies vom Netscape Navigator interpretiert wird. Im ersten Schritt wird hingegen eine Syntax integriert, die eine Ereignisüberwachung auch für den Internet Explorer gestattet. Mehr zu diesem Thema finden Sie im Abschnitt 4.9.2. Tritt innerhalb der aktuellen Seite das Ereignis **MOUSEDONW** ein, wird die Funktion `rechts()`

aufgerufen. Im nächsten Schritt wird die gleiche Funktion selbst aufgerufen und gibt beim gleichen Anwenderereignis die gleiche Meldung aus. Es lassen sich auch mehrere Ereignisse überwachen. Hierzu müssen der Syntax lediglich ein zweites oder auch weitere Ereignisse zugewiesen werden. Soll also beispielsweise neben dem **MOUSEDOWN** auch das **DBLCLICK**-Ereignis überwacht werden, muss die Syntax folgendermaßen modifiziert werden.

```
window.captureEvents(Event.MOUSEDOWN | Event.DBLCLICK )
```

Die einzelnen Ereignisse müssen im Zusammenhang mit einer Mehrfachüberwachung jeweils durch einen Querstrich voneinander getrennt werden. Die Anzahl der zu überwachenden Ereignisse spielt indes keine Rolle.

Überwachung beenden

eine etwas seltsam anmutende Syntax

Die **releaseEvents()**-Methode ist stets im festen Kontext mit der **captureEvents()**-Methode zu betrachten. So werden die innerhalb der **captureEvents()**-Methode definierten Anwenderereignisse, die überwacht werden sollen, durch den Einsatz der **releaseEvents()**-Methode nicht mehr überwacht. Es findet also eine Aufhebung der Überwachung innerhalb des definierten Fensters statt. Die Art der Ereignisse, deren Überwachung aufgehoben werden kann, ist die gleiche und erfordern demzufolge auch die gleiche Syntax, wie bei **captureEvents** demonstriert wurde. Die **releaseEvents()**-Methode erwartet als Parameter das oder die Ereignisse, deren Überwachung aufgehoben werden soll. Für den Fall, dass mehrere Ereignisse notiert werden sollen, müssen diese Event-Namen durch einen Querstrich voneinander getrennt notiert werden. Soll also beispielsweise die Überwachung der beiden Anwenderereignisse Klick und Doppelklick aufgehoben werden, müsste **Event.CLICK | EventDBLCLICK** notiert werden. Das folgende Beispiel zeigt eindrucksvoll den Einsatz der beiden Methoden **captureEvents()** und **releaseEvents()**. Innerhalb des Dateikörpers wird ein normaler Textabsatz spezifiziert. Klickt der Anwender auf diesen, hält die Maustaste gedrückt und bewegt den Cursor, bewegt sich das gesamte Browserfenster mit. Das Bewegen des Fensters wird hierbei durch die beiden Ereignisse **Event.MOUSEMOVE** und **Event.MOUSEDOWN** ausgelöst. Diese beiden Ereignisse werden durch die Definition dieser beiden Events innerhalb der **captureEvents()**-Methode überwacht. Innerhalb der Funktion `EndeBewegung()` wird diese Überwachung für das Ereignis **Event.MOUSEMOVE** wieder aufgehoben. Lässt der Anwender die gedrückte Maustaste wieder los, tritt also das Event **MOUSEUP** ein, löst dies die eben erwähnte Funktion `EndeBewegung()` aus und die Überwachung des **MOUSEMOVE**-Events wird aufgehoben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var links, oben;
function Bewegung(Koordinaten) {
   document.captureEvents(Event.MOUSEMOVE);
   document.onmousemove=AnfangBewegung;
   links=Koordinaten.pageX;
   oben=Koordinaten.pageY;
   return false;
}
function EndeBewegung(Koordinaten) {
   document.onmousemove=0;
   document.releaseEvents(Event.MOUSEMOVE);
   return false;
}
function AnfangBewegung(Koordinaten) {
   moveBy(Koordinaten.pageX - links, Koordinaten.pageY - oben);
   links = Koordinaten.pageX;
   oben = Koordinaten.pageY;
}
document.captureEvents(Event.MOUSEUP|Event.MOUSEDOWN);
document.onmousedown=Bewegung;
document.onmouseup=EndeBewegung;
// -->
</SCRIPT>
</HEAD>
</BODY>
<P>beweg mich</P>
</BODY>
</HTML>
```

Listing 4.44: Eine alternative Variante, den Browser zu bewegen

Beachten Sie, dass sich die gezeigte Syntax lediglich innerhalb des Netscape Navigators in den Produktversionen 4.x nachvollziehen lässt. Dass der Internet Explorer mit dem hier gezeigten Script nicht umgehen kann, ist zu Beginn dieses Abschnitts bereits erwähnt worden. Gleiches gilt jedoch auch für den Netscape Navigator in der Version 6.2.

4.9.4 Das Event-Objekt

sowohl im Internet Explorer als auch im Netscape Navigator verfügbar

Mit dem **event**-Objekt können Sie Informationen über Anwenderereignisse einholen. Diese Informationen können sehr genau sein. So lässt sich beispielsweise nicht nur ermitteln, ob ein Anwender einen Mausklick ausgeführt hat. Durch das **event**-Objekt kann zusätzlich die exakte Position des Mausklicks in Erfahrung gebracht werden. Zunächst die gute Nachricht. Der Internet Explorer und der Netscape Navigator kennen das **event**-Objekt. Nun zur schlechten. Die Unterschiede in der Implementierung sind gravierend. Nicht nur, dass die Syntax der Ereignisüberwachung eine jeweils andere ist. Zusätzlich hierzu existieren für beide Browser unterschiedliche Eigenschaften des **event**-Objekts. Diese werden auch lediglich von dem jeweiligen Browser unterstützt.

Sondertasten(Microsoft)

Nach Microsoft-Syntax lässt sich überprüfen, ob zusätzlich zu einer gedrückten Taste oder einem Mausklick eine der drei folgenden Sondertasten gedrückt wird. Tritt dieser Fall ein, lässt sich hierauf reagieren.

- **altKey** – bezeichnet die Alt-Taste
- **ctrlKey** – bezeichnet die Ctrl-Taste
- **shiftKey** – bezeichnet die ⇧-Taste

Für einen, wenn auch zugegebenermaßen nicht vollkommenen Kopierschutz für Elemente, der über die Tastenkombination Strg und C vorgenommen werden soll, kann die hier gezeigte Variante eine simple Möglichkeit bieten. Im folgenden Beispiel erhält der Anwender bei jedem Versuch, die Maustaste und gleichzeitig die ⇧-Taste zu drücken ein Meldefenster, in dem ein entsprechender hierauf abzielender Text angezeigt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JScript" for="document"
event="onclick()">
{
if(window.event.shiftKey)
alert("Sie haben zusätzlich die Shift-Taste gedrückt");
}
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.45: Die ⇧-Taste wird erkannt.

Es ist in jedem Fall darauf zu achten, dass ein alleiniges Notieren der **event.shiftKey-Angabe** nicht ausreicht. Vielmehr muss innerhalb des einleitenden **<SCRIPT>**-Tags das entsprechende Event, in diesem Fall ist dies **onKeypress,** exakt spezifiziert werden.

Pixel speichern (Microsoft)

Seitenbereiche verschieben

Nach Microsoft-Syntax lässt sich die Position des Cursors pixelgenau ermitteln und mit dem Eintreten eines definierten Ereignisses für gezielte Anwendungen weiterverarbeiten. So bietet diese Variante beispielsweise die Möglichkeit, bestimmte Bereiche einer Seite auf Grund eines eintretenden Ereignisses zu verschieben. Die zwei folgenden Anweisungen sind für die Positionsbestimmung des Cursors notwendig.

- **clientY** – speichert die vertikalen Pixel
- **clientX** – speichert die horizontalen Pixel

Wie sich diese Schlüsselwörter einsetzen lassen, soll anhand des nachstehenden Beispiels gezeigt werden. Als Event-Handler wurde hier **onmousedown** gewählt. Das Ereignis, auf welches reagiert wird, ist demnach also das Drücken der Maustaste innerhalb des Fensters. Innerhalb dieses Fensters wird ein **<DIV>**-Bereich mit einigen diesen beschreibenden Style-Sheet-Anweisungen notiert.

```
<HTML>
<HEAD>
<SCRIPT for="document" event="onmousedown( )" type="text/JScript">
<!--
{
document.all.verschieben.style.top = window.event.clientY;
document.all.verschieben.style.left = window.event.clientX;
}
//-->
</SCRIPT>
</HEAD>
</BODY>
<DIV id="verschieben" style="width:100; height:200; background-color:red;
position:absolute; left: 50; top:100">
Verschieb mich!!!</DIV>
</BODY>
</HTML>
```

Listing 4.46: Der <DIV>-Bereich kann verschoben werden.

Klickt nun der Anwender mit der Maus auf einen Bereich innerhalb des aktuellen Fensters, wird das **<DIV>**-Element, und somit alle hierin enthaltenen Elemente, an diese Stelle verschoben. Anhand dieser doch recht

simplen Syntax wird deutlich, wie sich mittels **event** phantastisch anmutende Ergebnisse erzielen lassen. Da eine vergleichbare Anwendung, wenn auch freilich unter Zuhilfenahme anderer Syntaxvarianten, innerhalb des Netscape Navigators möglich ist, lassen sich für eine solche Anwendung problemlos Crossbrowser-Lösungen umsetzen.

Dezimalcode speichern (Microsoft)

Nachrichten kodieren

Nach Microsoft-Syntax lässt sich durch den Einsatz des Jscript-Schlüsselworts **keyCode** der dezimale Wert einer gedrückten Taste ermitteln. Eingesetzt werden kann dieses Ereignis beispielsweise zur Kodierung von Nachrichten. Im folgenden Beispiel wird über ein Meldungsfenster der dezimale Code jeder gedrückten Taste ausgegeben. Ausgeführt wird **keyCode** in diesem Beispiel jedoch erst dann, wenn das Ereignis **onKeypress**, also das Drücken einer Taste, eintritt.

```
<SCRIPT for="document" event="onKeypress( )" type="text/JScript">
<!--
{
alert(window.event.keyCode);
}
//-->
</SCRIPT>
```

Eine korrekte Ausgabe des entsprechenden Dezimalcodes erfolgt immer dann, wenn das durch das gezeigte Script ausgezeichnete Fenster aktiv ist und ein Tastendruck erfolgt. Demzufolge sollte mit einer solchen Anwendung auf herkömmlichen Seiten eher sparsam umgegangen werden. Bleibt doch der praktische Nutzen einer solchen Syntax fraglich.

Dezimalcode (Netscape)

ähnlich wie **keyCode**

Vergleichbar ist **which** mit der im vorangegangen Abschnitt gezeigten **keyCode**-Anweisung des Internet Explorers. Unterschiede ergeben sich, neben den fast selbstverständlichen Syntaxunterschieden, vor allem auf Grund der diffizileren Handhabung der **which**-Anweisung. Zwar kann auch hierbei bei einem eintretenden Tastaturereignis der Dezimal-Code der entsprechenden Taste gespeichert werden, es existiert allerdings noch eine zweite Anwendungsform. Und zwar kann durch den Einsatz von **which** ermittelt werden, welche Maustaste vom Anwender gedrückt wurde. Und eben eine solche Anwendung, also die Überwachung der Maustaste, soll exemplarisch vorgeführt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
```

```
function Taste(welche)
{
alert("Sie haben die " + welche.which +". Maustaste gedrückt");
}
document.onmousedown = Taste;
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.47: Welche Maustaste wurde gedrückt?

Im gezeigten Beispiel wird die Mausaktivität überwacht. Klickt der Anwender eine Maustaste an, wird dies in einem Meldungsfenster ausgegeben. Hierbei werden numerische Werte angezeigt, wobei *1* das Anklicken der linken und *2* das Anklicken der rechten Maustaste bedeutet. Besitzt die Maus drei Tasten, verschieben sich diese Prioritäten. So hat in diesem Fall die linke, wie gehabt, den Wert *1*, die mittlere den Wert *2* und die rechte den Wert *3*.

Cursorposition (Microsoft)

Objekte pixelgenau positionieren

Die exakte Bestimmung der Cursorposition kann in vielerlei Hinsicht sinnvoll sein. Zum einen wird diese Anwendung sicherlich grafisch interessierten Programmierern ein geeignetes Werkzeug für die exakte Festlegung von Koordinaten innerhalb einer Grafik ermöglichen. Aber auch für andere Anwendungen bietet die hier gezeigte Syntaxform enormen Handlungsspielraum. So lässt sich beispielsweise genau ermitteln, an welcher Position sich ein bestimmtes Element befindet. Was wiederum für die Positionierung von Objekten, und dies gilt insbesondere im Hinblick auf verschiedene Browserversionen, sinnvoll erscheint. Nach Microsoft-Syntax stehen für die Ermittlung der Cursorposition zwei Schlüsselwörter zur Verfügung:

- **`offsetX`** – speichert die horizontalen Pixel
- **`offsetY`** – speichert die vertikalen Pixel

Zu berücksichtigen ist, dass die Cursorposition relativ zur linken oberen Ecke des Elements berechnet wird, von welchem das Ereignis ausgeht. Hierauf ist bei den erhaltenen Werten explizit zu achten. Es wird also nicht die Cursorposition zur linken oberen Ecke des Fensters, sondern zur linken oberen Ecke des Elements gespeichert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JScript">
```

```
function Koordinaten()
{
alert("Der Cursor befindet sich exakt: " + window.event.offsetX + " Pixel von
links und " +window.event.offsetY+" Pixel  von
oben");
}
</SCRIPT>
</HEAD>
</BODY>
<P  style="position:absolute; width:300; height:400; background-
color:blue;"onclick="Koordinaten()">
Die aktuelle Cursorposition</P>
</BODY>
</HTML>
```

Listing 4.48: Die Koordinaten des Cursors werden in einem Meldungsfenster ausgegeben.

In dem gezeigten Beispiel wurde durch das HTML-Tag **<P>** ein Textabsatz definiert. Dieser ist unter Zuhilfenahme diverser Style-Sheet-Anweisungen absolut Positioniert und mit einer festen Größe sowie einer Hintergrundfarbe ausgezeichnet. Klickt nun ein Anwender mit der Maus auf diesen Textabschnitts, wird die aktuelle Cursorposition innerhalb eines Meldungsfensters ausgegeben. Für die Ausgabe dieser Information wird die Funktion `Koordinaten( )` aufgerufen, und zwar mittels des **onclick**-Event-Handlers.

Cursorposition (Netscape)

Phantastische Einsatzmöglichkeiten im Hinblick auf die Bestimmung der Cursorposition bietet auch die in diesem Abschnitt vorzustellende Syntaxform. So lassen sich neben der simplen Erkennung der Cursorposition und deren Ausgabe in einem Meldungsfenster oder innerhalb der Statuszeile eben auch optisch ansprechende und grazil anmutende Anwendungen erstellen. Zunächst hier die zwei für die Erkennung der Cursorposition nach Netscape-Syntax notwendigen Schlüsselwörter:

- **pageX** – speichert die horizontalen Pixel
- **pageY** – speichert die vertikalen Pixel

Die Cursorposition wird stets relativ zur linken oberen Ecke der Seite verstanden. Eine eindrucksvolle Anwendungsvariante beschreibt das folgende Beispiel. In diesem wird die Funktion `Verschieber()` definiert. Ausgelöst wird diese durch das Drücken der Maustaste seitens des Anwenders, also durch den Event-Handler **onclick**. Als Bereich, auf welchen die erhaltenen Informationen angewandt werden sollen, wird hier, in typischer Netscape-4-Manier, ein Layer definiert. Durch zusätz-

liche Angaben wird diesem sein, wenngleich auch wenig graziöses, Erscheinungsbild zugewiesen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Verschieber(Richtung)
{
document.layers[0].top = Richtung.pageY;
document.layers[0].left = Richtung.pageX;
}
document.onmouseup = Verschieber;
</SCRIPT>
</HEAD>
</BODY>
<layer width="200" height="200" top="10" left="70"  bgcolor="yellow">Bitte
anklicken</layer>
</BODY>
</HTML>
```

Listing 4.49: Der Layer folgt dem Cursor.

Klickt der Anwender innerhalb des ausgewiesenen Layers mit der Maus, passiert zunächst gar nichts. Denn wirklich interessant wird dieses Beispiel erst in dem Moment, wenn der Nutzer mit der Maus außerhalb dieses Bereichs eine Fläche der Seite anklickt und somit die Funktion `Verschieber(Richtung)` auslöst. Tritt dieses Ereignis ein, wird der gesamte Layer an diese Position verschoben. Die Position dieses Verschiebens orientiert sich an dem exakten Auftreten des Cursors und dessen Koordinaten. Die neue Position des Layers beginnt dann exakt an diesem Punkt, und zwar an dessen linker oberer Ecke.

Größe von Objekten (Netscape)

zwei Einsatzvarianten

Die hier gezeigten Schlüsselwörter stehen nach Netscape-Syntax für zwei unterschiedliche Zwecke zur Verfügung. Die Unterscheidung, für welchen Einsatzzweck sie verwendet werden, ergibt sich aus der Zuweisung unterschiedlicher Event-Handler. Aus diesem Grund ist eine Unterscheidung zwischen den zwar gleich bleibenden Schlüsselwörtern und deren variierender Bedeutung unumgänglich und soll hier vorgenommen werden. Für das Ereignis **`onresize`** ergibt sich die folgende Bedeutung für den Einsatz dieser Schlüsselwörter. Dieses Ereignis tritt dann in Kraft, wenn die Größe des Browserfensters verändert wird.

- **`layerX`** – speichert die Breite des Objekts
- **`layerY`** – speichert die Höhe des Objekts

Eine weitere Einsatzmöglichkeit besteht in der Ermittlung der Cursorposition. Der Netscape Navigator berechnet diese anhand der linken oberen Ecke des aktuellen Layer-Bereichs. Für diese Anwendung können alle bekannten Event-Handler, aber eben nicht das Ereignis **onresize**, eingesetzt werden. Für diesen Fall besitzen die beiden Schlüsselwörter die folgende Bedeutung:

- **layerX** – speichert die vertikalen Pixel
- **layerY** – speichert die horizontalen Pixel

Im folgenden Beispiel wird durch die Funktion `Groesse()` das Ereignis **onresize** überwacht. Tritt dieses Ereignis ein, also verändert der Anwender die Größe des Browserfensters, wird dessen neue Größe innerhalb der Statuszeile angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Groesse(Pixel)
{
window.status = " Breite: " + Pixel.layerX + " Höhe: " + Pixel.layerY;
return true;
}
window.onresize = Groesse;
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Listing 4.50: Die neue Größe des Browserfensters wird ermittelt.

Die aktuelle Größe der Browserinstanz wird erst dann angezeigt, wenn der Vorgang der Größenänderung abgeschlossen ist. Während der Veränderung der Anzeigengröße werden in der Statuszeile indes keine Pixelangaben angezeigt.

Zusatztasten (Netscape)

nicht auf jeder Tastatur verfügbar

Auch Netscape bietet, ähnlich wie Microsoft, eine Syntax für die Überprüfung von gedrückten Zusatztasten an. Hierbei ist natürlich zunächst zu bedenken, dass sich diese Tasten selbstverständlich nicht auf jeder Tastatur befinden und somit Scripts, die speziell hierauf aufbauen, im Normalfall vermieden werden sollten. Laut Netscape-Syntax erfolgt die Abfrage der jeweiligen Tasten über die folgenden vier Anweisungen:

- name.**modifiers&Event.ALT_MASK** – fragt die Alt-Taste ab
- name.**modifiers&Event.CONTROL_MASK** – fragt die Strg-Taste ab
- name.**modifiers&Event.SHIFT_MASK** – fragt die ⇧-Taste ab
- name.**modifiers&Event.META_MASK** – fragt die AltGr-Taste ab

name darf frei gewählt werden, da es sich hierbei um ein Element der Funktion handelt, welches vom Programmierer gesetzt wird. Auf die Schreibweise ist exakt zu achten, da der Netscape Navigator hier Groß- und Kleinschreibung explizit unterscheidet und bei Missachtung der gezeigten Schreibweise die Verarbeitung des JavaScript-Programms nicht fortsetzen kann. Exemplarisch für die Abfrage der Zusatztaste Strg steht die nachfolgende Syntax zur Verfügung.

```
<HTML>
<HEAD>
<SCRIPT type="textJavaScript">
function taste(welche)
{
if(welche.modifiers & Event.CONTROL_MASK)
alert("Warum drücken Sie die Strg-Taste?")
}
document.onmouseup = taste;
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Angezeigt wird das Meldungsfenster mit dem Hinweistext jedoch erst, wenn die Zusatztaste parallel zu einer anderen Taste oder einem Mausklick gedrückt wird. So würde in diesem Beispiel das alleinige Drücken der Strg-Taste zu keinem Anzeigen des Meldungsfensters führen.

Art des Ereignisses (Netscape)

Event-Handler als Rückgabewert

Nach Netscape-Syntax kann unter Verwendung des **type**-Events die Art des eintretenden Ereignisses abgefragt werden. Als Wert liefert **type** den entsprechenden Event-Handler zurück. Bei der Ausgabe des Ereignisses ist eine Netscape-spezifische Schreibweise zu beachten. Alle Event-Handler werden hier nämlich ohne das Präfix **on** angezeigt. So wird innerhalb des Netscape Navigators der Event-Handler **onKeypress** beispielsweise zu **keypress**. Zu beachten ist, dass sich diese etwas anders geartete Schreibweise einzig und allein auf die Ausgabe bezieht. Innerhalb des eigentlichen Scripts muss nach wie vor die herkömmliche Syntax verwendet werden. Exemplarisch soll die Verwendung des **type**-Schlüsselworts an-

hand der nachstehenden Syntax veranschaulicht werden. Hier werden drei mögliche Ereignisse spezifiziert. Tritt eines von diesen innerhalb des aktuellen Fensters ein, wird das entsprechende Ereignis, eingebettet in einen etwas blumig anmutenden Text, innerhalb eines Meldungsfensters ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Ereignis(Art)
{
alert("Sie bedienen sich des " + Art.type + " Event-Handlers");
}
document.onkeyup = Ereignis;
document.onmouseup = Ereignis;
document.onKeypress = Ereignis;
</SCRIPT>
</HEAD>
</BODY>
</BODY>
</HTML>
```

Sinnvoll ist der Einsatz des **type**-Schlüsselworts vor allem dann, wenn abgefragt werden soll, welches Ereignis durch den Nutzer eingeleitet wurde. Das gezeigte Beispiel lässt sich mit allen einsetzbaren Event-Handlern, die der Netscape Navigator kennt, vervollständigen. An der eigentlichen Funktion muss nichts verändert werden, da jedes neu definierte Ereignis automatisch die Funktion `Ereignis()` aufruft.

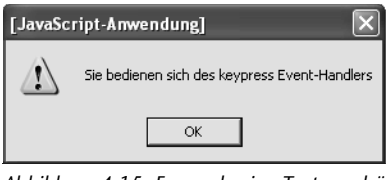


Abbildung 4.15: Es wurde eine Taste gedrückt.

Bildschirmkoordinaten (Netscape)

Position des Cursors

Laut Netscape-Syntax lässt sich die Position des Cursors bestimmen, wenn ein Ereignis eintritt. Die Anwendungsmöglichkeiten erstrecken sich auch hierbei von rein grafischen bis hin zu programmiertechnischen innovativen Lösungen. Die beiden für den Zweck der Positionsbestimmung von Netscape favorisierten Schlüsselwörter lauten folgendermaßen:

- **screenX** – speichert die horizontalen Pixel
- **screenY** – speichert die vertikalen Pixel

Die aus dem Einsatz dieser Schlüsselwörter resultierenden Pixel-Ergebnisse orientieren sich an den jeweiligen Fensterrändern. So hat die Anweisung **screenX** zur Folge, dass die Pixel vom oberen Fensterrand gezählt werden, wohingegen die Angabe **screenY** die Cursorposition vom linken Fensterrand aus speichert. Die Anzeige der Cursorposition sollte im Normalfall mit einem Event-Handler im Kontext mit einem geeigneten Ereignis realisiert werden. So bietet sich beispielweise der Einsatz des Event-Handlers **onclick** nahezu an, wohingegen **onkeydown** wenig erbaulich sein dürfte. Um dieser Aussage Nachdruck zu verleihen, ist der die Funktion `Ereignis()` auslösende Event-Handler **onclick**.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Ereignis(Art)
{
alert(Art.screenX + " Pixel von links " + Art.screenY + "
Pixel von oben ");
}
document.onclick = Ereignis;
</SCRIPT>
</HEAD>
</BODY>
<P>&Uuml;berpr&uuml;fen Sie Ihre aktuelle Position!!!</P>
</BODY>
</HTML>
```

Tritt das Ereignis **onclick** ein, wird die Position des Cursors innerhalb eines über **alert( )** realisierten Ausgabefensters angezeigt. Es sei an dieser Stelle nochmals darauf hingewiesen, dass für eine solche Anwendung tatsächlich nur Mausereignisse wie eben beispielsweise **onclick** verwendet werden sollten, um auch tatsächlich ein schlüssiges Ergebnis zurückzuliefern. Denn gesetzt den Fall, man würde auf ein Tastaturereignis setzen, hätte die Cursorposition wohl reinen Makulaturcharakter.

Bildschirmkoordinaten (Microsoft)

Mausereignisse verwenden

Laut Microsoft-Syntax kann die Cursorposition ausgelesen werden. Diese Möglichkeit sollte stets im Zusammenhang mit einem Mausereignis gesehen werden. Denn nur wenn tatsächlich hierauf eingegangen wird, macht die Koordinatenerkennung Sinn. Verzichtet werden sollte hingegen auf Tastaturereignisse. Die beiden Werte für das Auslesen von Bildschirmkoordinaten lauten nach Microsoft-Syntax folgendermaßen:

- x – speichert die horizontalen Pixel
- y – speichert die vertikalen Pixel

Als Bezugspunkt für die entsprechenden Pixelangaben, die der Internet Explorer zurückliefert, ist die linke obere Ecke des Elternelemenents bzw. die des Browserfensters spezifiziert. Auf das Browserfenster wird immer dann Bezug genommen, wenn sich das Element, welches die entsprechende Funktion auslöst, nicht innerhalb eines Elternelements befindet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JScript">
function Koordinaten()
 {
window.status = "x = " + window.event.x + "/ y = " + window.event.y;
 }
</SCRIPT>
</HEAD>
</BODY>
<DIV style="position:absolute; top:30; left:100; background-
color:blue;"onmouseover="Koordinaten()">
Achten Sie auf die Statuszeile</DIV>
</BODY>
</HTML>
```

In dem gezeigten Beispiel wurde ein **<DIV>**-Bereich, der mit Hilfe von Style-Sheet-Angaben sein spezielles Aussehen und die Positionsinformationen erhält, spezifiziert. Tritt das Ereignis **onmouseover** in Kraft, also fährt der Nutzer mit der Maus über den **<DIV>**-Bereich, wird die Funktion `Koordinaten()` ausgelöst. Diese zeigt die Koordinaten der aktuellen Cursorposition innerhalb der Statuszeile des Internet Explorers an.

4.9.5 Crossbrowser-Lösung

Vorüberlegungen

Wie Sie den bisherigen Ausführungen entnehmen konnten, verhalten sich der Internet Explorer und der Netscape Navigator bezüglich der Ereignisbehandlung sehr unterschiedlich. Nachdem wir die unterschiedlichen Konzepte kennen gelernt haben, müssen wir diese so umsetzen, dass sie sich in beiden Browsern nutzen lassen. Es existieren nun freilich unterschiedliche Herangehensweisen, um die beiden Browser zu unterscheiden. Wir werden hier zwei verschiedene Wege gehen. Um ein Script zu erstellen, welches ausschließlich vom Internet Explorer interpretiert wird, bedienen wir uns einer bekannten Tatsache. Jeder Browser interpretiert nur solche Script-Sprachen, die er kennt. Um dem Browser die verwendete Script-Sprache mitzuteilen, wird dem **type**-Attribut die verwendete Sprache zugewiesen. Da ausschließlich der Internet Explorer die Sprache Jscript interpretieren kann, weisen wir dem **type**-Attribut

den Wert **text/jscript** zu. Somit ist sichergestellt, dass dieser Bereich ausschließlich vom Internet Explorer interpretiert wird. Andere Browser ignorieren diesen. Da der Internet Explorer und der Netscape Navigator JavaScript-Bereiche interpretieren, muss für den Navigator eine andere Lösung gefunden werden. Um den Netscape Navigator zu erkennen, kann man sich der **appName**-Eigenschaft des **navigator**-Objekts bedienen. Diese Eigenschaft liefert als Rückgabewert den Namen des verwendeten Browsers. Es muss also lediglich der Rückgabewert dahingehend überprüft werden, ob dieser `Netscape` ist. Im folgenden Beispiel wird kontrolliert, ob der Anwender eine Taste gedrückt hat. Ist dies der Fall, wird das Ereignis **keydown** in die Statuszeile geschrieben.

```
<HTML>
<HEAD>
<SCRIPT for="document" event="onkeydown()" type="text/jscript">
<!--
{
window.status = "Ereignis " + window.event.type ; return true;
}
//-->
</SCRIPT>
<SCRIPT type="text/JavaScript">
<!--
var Netscape = new Boolean();
if(navigator.appName == "Netscape")  Netscape = true;
function gedrueckteTaste(Ereignis)
{
 if(Netscape)
  { window.status = "Ereignis " + Ereignis.type ; return true; }
}
document.onkeydown = gedrueckteTaste;
//-->
</SCRIPT
</HEAD>
<BODY>
</BODY>
</HTML>
```

Der Jscript-Bereich im oberen Teil des Programms wird ausschließlich vom Internet Explorer interpretiert. Somit können dort alle Explorerrelevanten Angaben notiert werden. Um den Typ des Ereignisses zu ermitteln, wird hier die Anweisung **window.event.type** notiert. Die Ausgabe des ermittelten Wertes wird durch **window.status** in der Statuszeile angezeigt. Im zweiten Script-Bereich, der ja ausschließlich für den Netscape Navigator bestimmt ist, muss zunächst garantiert sein, dass dieser auch ausschließlich vom Navigator ausgeführt wird. Hierzu wird die Variable

Netscape deklariert. Als Rückgabewert wird einer der boolschen Werte **true** oder **false** erwartet. Ist dieser Wert **true,** handelt es sich bei dem verwendeten Browser um den Netscape Navigator. Hieraufhin wird überprüft, ob der Anwender die Taste drückt. Hierzu dient die Anweisung **document.onkeydown = gedrückte**. Dadurch wird die Funktion gedrueckteTaste() aufgerufen. Diese schreibt den Typ des Ereignisses, also das Drücken der Taste, in die Statuszeile.

4.10 Entities

Sonderzeichen deklarieren

Unter Entities sind JavaScript-Anweisungen zu verstehen, welche innerhalb von HTML-Tags als Attribut notiert werden können. Da in der eigentlichen Syntax-Form an dieser Stelle lediglich Text integriert werden dürfte, besteht die Pflicht, die gewünschten JavaScript-Anweisungen in einer gesonderten Form wiederzugeben. Die JavaScript-Spezifikation bedient sich in einem solchen Fall der Notation für Sonderzeichen. Um Verwechslungen zwischen JavaScript-Entities und HTML-Entities zu vermeiden, müssen Entities in JavaScript in geschweifte Klammern eingeschlossen werden. In der allgemeinen Form stellt sich die Syntax eines JavaScript-Entity folgendermaßen dar:

```
<element attribut="&{
javascript-anweisung };">Inhalt</element>
```

Es ist zwingend darauf zu achten, dass JavaScript-Entities nur innerhalb von HTML-Tags verwendet werden dürfen. Deren Aufruf und Ausführung wird mit dem Laden der Seite eingeleitet. Alle gespeicherten Daten stellen Strings dar, die im Anschluss an den Aufruf ausgegeben werden. Eine wenn auch zugegebenermaßen nicht sehr sinnvolle Anwendung eines JavaScript-Entity beschreibt das folgende Beispiel.

```
var Farbe = "#000000";
<BODY bgcolor="&{Farbe};">
```

Die aufgeführte Variable Farbe dient hier nur der Veranschaulichung der Anwendung. Nach dem Aufruf der Seite wird die Variable Farbe, welcher ein hexadezimaler Farbwert zugewiesen wurde, ausgeführt. Deren Notation bedient sich innerhalb des **<BODY>**-Tags der JavaScript-Entity-Syntax und wird dem Attribut **bgcolor** als Wert zugewiesen.

4.11 Programmierstil

Programmierstil ist Geschmackssache!

Die Überschrift dieses Kapitels impliziert es bereits: Was hier als Programmierstil benannt ist, hängt in vielerlei Hinsicht vom Geschmack des Entwicklers ab. Dennoch sollen einige grundlegende Möglichkeiten genannt werden, mit deren Hilfe sich gut lesbare JavaScript-Programme entwickeln lassen. Die hier beschriebenen Regeln sind nicht als starre Richtlinien zu verstehen. Selbstverständlich werden Sie bei einigen Problemen andere Kriterien als die genannten anwenden. In diesem Kapitel werden so genannte Kann- und Muss-Regeln vorgestellt. Die Muss-Regeln sollten in jedem Fall eingehalten werden, da deren Missachtung zu Problemen und Laufzeitfehlern führen kann. Unter den Kann-Regeln sind all diejenigen Hinweise zusammengetragen, die sich aus meiner Erfahrung bewährt haben. Diese sind, wie bereits erwähnt, auf Grund der unterschiedlichen Geschmäcker nicht jedermanns Sache, sie ermöglichen aber ein auf Dauer effizienteres Arbeiten sowie eine schnellere Erlernbarkeit von JavaScript.

Dass dem Programmierstil hier ein eigenes Kapitel gewidmet ist, mag einige Leser verwundern. Schließlich funktionieren zumeist auch solche JavaScript-Programme, bei denen die hier beschriebenen Konventionen nicht eingehalten wurden. Und dennoch: JavaScript-Programme können nach der Implementierung und anschließender Testphase kaum über einen längeren Zeitraum unverändert bleiben. Dies liegt an der fortschreitenden Browsertechnologie ebenso wie an den stetig steigenden Kundenbedürfnissen. Demzufolge sollte innerhalb eines Unternehmens stets nach einheitlichen Kriterien programmiert werden. Nur so ist es ohne größeren Zeitverlust möglich, dass Programme im Nachhinein auch von anderen Personen als dem Entwickler selbst modifiziert werden können. Aber auch für den Programmierer selbst gilt, dass die Nichteinhaltung grundlegender Konventionen dazu führt, dass komplexe Syntaxformen häufig nicht schnell genug entschlüsselt und die entsprechend zu ändernden Passagen nicht gefunden werden. Als Beispiel soll die folgende Funktion dienen. Diese ist zwar im Umfang nicht groß, aber dank des schlechten Programmierstils nur schwerlich lesbar. Dass sich das Unverständnis bei einem umfangreicheren JavaScript-Programm summiert, ist leicht vorstellbar.

```
function hallo() {alert("Hallo Welt};}
```

Zugegebenermaßen handelt es sich hierbei um eine kleine Funktion, die in platzsparender Art und Weise programmiert wurde. Ein JavaScript-Programm von beispielsweise zehn Seiten in diesem Stil zu entschlüsseln ist da schon schwerer. Das folgende Syntaxfragment zeigt die gleiche Funktion. In diesem Beispiel wurde jedoch auf Konventionen wie Einrückungen usw. geachtet.

```
function halloWelt()
{
   alert("Hallo Welt");
}
```

Regeln konsequent einhalten

Als Kriterium sei an dieser Stelle Folgendes herausgestellt: Je umfangreicher das Programm, desto gewissenhafter sollte sauberer Programmierstil angewandt werden. Dass in diesem Buch nicht fortlaufend die Konventionen eingehalten werden, liegt an zweierlei Dingen. Zum einen würden unnötige Seitenumbrüche entstehen und zum anderen würde der Umfang des Buches unverhältnismäßig zunehmen. Noch einige grundlegende Worte zum Thema Programmierstil. Betrachten Sie die meisten der hier vorgestellten Konventionen als Empfehlungen, die Ihnen das Arbeiten mit und das Erlernen von JavaScript erleichtern sollen. Wichtig ist nur eins. Sie sollten, wenn Ihnen einige Programmierstil-Hinweise nicht geheuer sind, einfach eigene definieren und diese dann ebenfalls konsequent einhalten. Nur so können Sie sicher sein, dass Sie die Programme auch in einem halben Jahr noch schnell bearbeiten können.

4.11.1 Programm-Layout

wichtige Grundregeln

Um Programme auch für andere Entwickler lesbar zu machen bzw. zu gewährleisten, dass das Programm auch in einigen Monaten noch schnell verstanden werden kann, sollten grundlegende Prinzipien der äußeren Programmform eingehalten werden. Auf alle nachfolgend aufgeführten Aspekte des Programm-Layouts wird im Laufe dieses Abschnitts noch spezifischer eingegangen. An dieser Stelle soll lediglich ein Eindruck über die wichtigsten Grundregeln vermittelt werden.

- Häufig werden Variablen- und Funktionsnamen in abgekürzter Schreibweise notiert. Das ist in vielerlei Hinsicht korrekt und hiergegen ist nichts einzuwenden. Es sollte jedoch darauf geachtet werden, dass die verwendeten Abkürzungen logisch und auch nach einem längeren Zeitraum noch entschlüsselbar sind.
- Um die Lesbarkeit eines Programms zu erhöhen, sollten in jedem Fall Einrückungen vorgenommen werden. Zwar kann hier nur vorgeschlagen werden, wie diese Einrückungen vorgenommen werden. Als wichtigstes Kriterium sollte jedoch stets die Lesbarkeit eines Programms verstanden werden. So sollten beispielsweise Anweisungsblöcke jeweils konstant um eine festgelegte Anzahl von Stellen eingerückt werden. Darüber hinaus ist darauf zu achten, dass öffnende und schließende geschweifte Klammern jeweils untereinander angeordnet sind.
- Das Nichtvorhandensein von Kommentaren zeugt von schlechtem Programmierstil und ist ein Qualitätsmangel! Nur durch sorgfältig eingefügte Kommentare lassen sich JavaScript-Programme schnell

lesen und lässt sich deren Verwendungszweck auf den ersten Blick erkennen.

- Ohne Leerzeilen fällt die optische Gliederung eines JavaScript-Programms schwer. Als Faustregel gilt, dass sowohl einzelne Anweisungsblöcke wie auch verschiedene Funktionen durch Leerzeilen getrennt werden sollten.

Beachten Sie, dass hier nicht alle Aspekte des guten Programmierstils berücksichtigt wurden. Viele Regeln werden Sie selbst erstellen oder auch auf einige verzichten. Dennoch gilt, dass sich nur durch das Einhalten gewisser stilistischer Feinheiten sich qualitativ hochwertige JavaScript-Programme entwickeln lassen.

4.11.2 Abkürzungen

Logische und gängige Namen wählen!

Abkürzungen sind ein hervorragendes Mittel, um lange Namen innerhalb von JavaScript-Programmen zu umgehen. Lange Namen sind Fehlerquellen und somit im Regelfall zu vermeiden. Zudem sind diese umständlich zu handhaben. Zugegebenermaßen ist es nicht ganz einfach, für einen langen Variablennamen eine Abkürzung zu finden. In vielen Fällen lassen sich etwaige Probleme jedoch ganz einfach lösen. So banal es klingen mag: Für eine Variable, welche die Stunde der aktuellen Uhrzeit ausgeben soll, lässt sich zwar der Variablenname *s* verwenden, logischer wäre hier jedoch *Std*. So simpel dieses Beispiel auch ist, es ist tatsächlich so, dass sich viele Wörter auf einprägsame Weise abkürzen lassen. Zudem sollten sich die verwendeten Abkürzungen am allgemeinen Sprachgebrauch orientieren. Auch das wird Ihnen später helfen, die verwendeten Abkürzungen wiederzuerkennen. Problematisch ist es, wenn mehrere Abkürzungen dazu führen, dass die Funktions- oder Variablennamen fast identisch sind. In einem solchen Fall kann beispielsweise ein kleiner Zusatz im Namen wahre Wunder bewirken. Als Faustregel für Abkürzungen gilt: Wortanfänge sind wichtiger als Wortenden. Nehmen wir das Szenario nach der Suche einer Abkürzung für den Begriff *Lampe*. Die Verwendung des Wortanfangs `Lam` ist hier bei weitem effizienter als der Einsatz des Wortendes *mpe*.

4.11.3 Einrückungen und Leerzeilen

Um eine übersichtliche Programmstruktur entwickeln zu können, ist das Stilmittel der Einrückungen ein unerlässliches Muss. Zur Veranschaulichung dieser These soll ein kurzer Ausflug in die HTML-Welt dienen. Eines der dort vorherrschenden Elemente sind Tabellen. Deren Gestaltung ist sicherlich nicht sonderlich schwierig. Und dennoch passiert es häufig, dass Tabellenkonstrukte fehlerhaft sind. Dies liegt weniger an

der Unwissenheit der Gestalter der Seiten als vielmehr daran, dass auf Einrückungen verzichtet wurde. Denn würden sich alle <TR>-Tags einer Tabelle untereinander befinden und würde dieses Vorgehen auch auf das <TD>-Tag angewandt werden, würde es bei weitem weniger Probleme geben. Das Gleiche lässt sich nun auch auf JavaScript-Programme anwenden. Nur sollten hier nicht Tags, sondern Anweisungen eingerückt werden. So hat es sich als effizient erwiesen, wenn geschweifte Klammern, die einen Anweisungsblock umgeben, jeweils auf die gleiche Art und Weise eingerückt werden. Wie weit Einrückungen erfolgen sollten, ist freilich wieder eine Stilfrage. Dennoch sollte als Kriterium gelten, dass Anweisungsblöcke jeweils zwei bis vier Stellen eingerückt werden.

Leerzeilen

Um JavaScript-Programme weiter gliedern zu können, empfiehlt sich der Einsatz von Leerzeilen. Hierdurch lassen sich Programmstrukturen hervorheben, ohne dass auf Hilfsmittel wie Sternenlinien usw. zurückgegriffen werden muss. Leerzeilen sollten in jedem Fall zwischen einzelnen Funktionen gesetzt werden. Aber auch Variablendeklarationen sollten von Anweisungsblöcken mittels Leerzeilen optisch getrennt werden.

4.11.4 Kommentare

qualitativ hochwertige Programme

Das Nichtvorhandensein von Kommentaren in einem JavaScript-Programm ist stets als Qualitätsmangel zu betrachten. Kommentare sollten nicht als bloßes Beiwerk, sondern als elementarer Bestandteil von JavaScript verstanden werden. Grundsätzlich unterscheiden wir zwei verschiedene Arten von Kommentaren. Zum einen existieren solche, welche die Funktion, die Version, aber auch den Autor des Programms beschreiben. Diese Kommentare sollten stets am Programmanfang notiert werden. Bei der anderen Art von Kommentaren handelt es sich um solche, die Objekte und Anweisungsblöcke beschreiben. Diese werden innerhalb des Anweisungsteils von JavaScript-Programmen notiert. Weitere Regeln für den Umgang mit Kommentaren beschreiben die folgenden Punkte:

- Umfangreiche Anweisungsblöcke sollten in jedem Fall mit Kommentaren versehen werden.
- Kommentare sollten immer direkt während der Programmierung notiert und nicht erst nachträglich eingefügt werden.
- Kommentare sollten stets kurz und prägnant formuliert werden. Eine Auflistung des Quellcodes in Satzform ist keine geeignete Art der Kommentierung.
- Im Idealfall werden Angaben zu den das Programm unterstützenden Browserversionen gemacht.
- Im Regelfall sollte auf Elemente wie Linien und Sterne zur Strukturierung von JavaScript-Programmen verzichtet werden.

Noch einige abschließende Bemerkungen: Nicht jedes JavaScript benötigt Kommentare, um verständlich zu sein. Dennoch zeigt die Erfahrung, dass Programme mit Kommentaren sich auch noch nach langer Zeit leichter modifizieren lassen als solche ohne Kommentare. Selbstverständlich kann das Prinzip des Kommentierens nur dann funktionieren, wenn die Kommentare direkt während der Programmierung und nicht erst im Nachhinein eingefügt werden.

4.12 Fragen und Übungen

1. Ein Programm soll den Text `Hallo Welt` in einem Meldungsfenster ausgeben. Welche(s) Code-Fragment(e) ist (sind) korrekt?

```
<SCRIPT type="text/JavaScript">
<!--
alert("Hallo Welt">
//-->
</SCRIPT>

<SCRIPT type="text/JavaScript">
<!--
document.write("Hallo Welt">
//-->
</SCRIPT>

<SCRIPT type="text/JavaScript">
<!--
var Text = "Hallo Welt";
alert("Text">
//-->
</SCRIPT>
```

2. Welcher der folgenden Event-Handler ist in JavaScript nicht bekannt?

onload
onmove
onTime

5 Fenster

lernen

Ziele dieses Kapitels

Unter dem Begriff Fenster wird in JavaScript das gesamte Browserfenster verstanden. Sie können mit dem in diesem Kapitel vermittelten Wissen beispielsweise neue Fenster öffnen und schließen. Ebenso werden Sie erfahren, wie sich die berühmten Vor- und Zurück-Buttons auf Internetseiten realisieren lassen. Zusätzlich erfahren Sie, wie Sie Dialogfenster generieren und Meldungsfenster für sinnvolle Anwendungen nutzen können. Beachten Sie, dass Sie den Inhalt dieses Kapitels in jedem Fall für die Erstellung von JavaScript-Programmen benötigen. Sie sollten also erst dann mit der Lektüre dieses Buches fortfahren, wenn Sie die Grundlagen der Arbeit mit Fenstern verinnerlicht haben.

5.1 Zugriff auf Fenster

das oberste Objekt der Objekthierarchie

Für alle Elemente, die innerhalb des Browserfensters angezeigt werden, ist das **window**-Objekt das oberste Objekt. Das **window**-Objekt ist eines der am häufigsten verwendeten JavaScript-Objekte. Lassen sich doch mit diesem Fenster abfragen und kontrollieren. Zudem wird es möglich, neue Fenster zu öffnen und deren Eigenschaften selbst zu bestimmen. So können beispielsweise neue Fenster ohne Menüleiste und Scrollbalken geöffnet werden. Folgende allgemein gültige Syntaxformen können für den Zugriff auf Fenster verwendet werden:

```
window.Eigenschaft
window.Methode()
```

Hinter dem **window**-Objekt wird eine der vordefinierten Eigenschaften bzw. Methoden notiert. Statt des **window**-Objekts können Sie auch das Schlüsselwort **self** verwenden. Da die Verwendung des **window**-Objekts jedoch weiter verbreitet ist, wird in diesem Buch ausschließlich dieses verwendet. Die nachstehende Syntax zeigt exemplarisch die Verwendung des **window**-Objekts im Zusammenhang mit Eigenschaften und Methoden.

```
window.innerHeight = "300";
window.alert("Hallo Welt");
```

Da es sich bei dem **window**-Objekt um das höchste in der Objekthierarchie handelt, ist dessen Notation nicht zwingend erforderlich. Die Anweisung **window.alert()** lässt sich folgendermaßen abkürzen:

```
alert("Hallo Welt")
```

Im folgenden Beispiel wird durch die Verwendung der **status**-Eigenschaft auf die Statuszeile des Browsers zugegriffen und in dieser ein Text angezeigt. Dies wird ausgeführt, wenn der Anwender mit der Maus über den Hyperlink fährt. Zu diesem Zweck wird der Event-Handler **onmouseover** verwendet.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="neu.htm" onmouseover="window.status='neue Seite';
return true;">weiter</A>
</BODY>
</HTML>
```

Listing 5.1: Ein beschreibender Text in der Statuszeile

Wie schon erwähnt, können Sie nicht nur das aktuelle Fenster ansprechen. In JavaScript ist es möglich, neue Fenster zu erzeugen. Auf diese neuen Fenster kann dann wiederum zugegriffen werden. So wird es beispielsweise möglich, andere Fenster aus einem Dokument heraus zu schließen oder dieses mit neuen Inhalten zu belegen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Zweitfenster()
{
Fenster= window.open("nuke.htm", "neu", "width=300,height=350");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Zweitfenster()">Fenster öffnen</A>
</BODY>
</HTML>
```

Listing 5.2: Ein neues Fenster wird geöffnet.

Durch Anklicken des Verweises wird die Funktion `Zweitfenster()` aufgerufen. Hierdurch wird ein neues Fenster geöffnet. Dieses Fenster wird in einer Breite von `300` und einer Höhe von `350` Pixel geöffnet. Als Inhalt wird die Datei `nuke.htm` angegeben. Um das Fenster ansprechen zu können, muss eine Variable erzeugt werden, in der die Fensterinstanz gespeichert wird. In diesem Beispiel ist dies `Fenster`. Über diesen Namen lässt sich das geöffnete Fenster ansprechen und auf dieses alle Eigenschaften und Methoden anwenden. Nehmen wir also an, dass das neue Fenster aus dem Hauptfenster heraus geschlossen werden soll. Dazu müsste die zuvor gezeigte Syntax um folgende Zeile ergänzt werden:

```
<A href="javascript:Fenster.close()">Fenster schließen</A>
```

Das neu erzeugte Fenster, dem zuvor der Name `Fenster` zugewiesen wurde, ist über diesen Namen ansprechbar. Um das Fenster zu schließen, wird dem Namen `Fenster` die Methode **`close()`** zugewiesen. Diese sorgt für das Schließen des Fensters.

Eine weitere Möglichkeit besteht darin, das Hauptfenster aus dem neu geöffneten Fenster heraus anzusprechen. Auch hierbei sind wieder alle Eigenschaften und Methoden verwendbar. Nehmen wir an, dass Hauptfenster soll aus dem neu geöffneten Fenster heraus geschlossen werden. Der Quellcode der Datei `nuke.htm`, also der Datei, die im Fenster `Fenster` angezeigt wird, muss hierfür folgendermaßen angepasst werden:

```
<A href="javascript:opener.close()">Hauptfenster schließen</A>
```

Um das Hauptfenster anzusprechen, wird das vordefinierte Fenster-Objekt **`opener`** verwendet. Notieren Sie hinter dem Schlüsselwort **`opener`**, durch einen Punkt getrennt, die gewünschte Eigenschaft oder Methode. Da in unserem Beispiel das Fenster geschlossen werden soll, wird die Methode **`close()`** verwendet.

5.2 Fenster öffnen

Die Methode **`open()`** ist wohl eine der am häufigsten eingesetzten JavaScript-Methoden und ist aus modernen Internetauftritten kaum noch wegzudenken. Dies mag zum einen an den zahlreichen Anwendungsmöglichkeiten liegen, zweifelsohne spielt aber auch die simple Syntax dieser Methode eine entscheidende Rolle für deren Erfolgsgeschichte. Der **`open()`**-Methode können zahlreiche Parameter zugewiesen werden. Gesetzt werden müssen in jedem Fall die beiden folgenden Parameter.

mögliche Parameter

- URL – Die Adresse der Datei, welche innerhalb des neu geöffneten Fensters angezeigt werden soll, muss exakt spezifiziert werden. Es gelten hierbei die gleichen Regeln wie für das Setzen von Verweisen in

HTML. Es findet also auch hierbei eine Unterscheidung zwischen relativen und absoluten Pfadangaben statt.

- Fenstername – Es handelt sich hierbei um ein entscheidendes Kriterium für die Weiterverarbeitung des neu geöffneten Fensters. Nur anhand eines eindeutig vergebenen Namens kann das Fenster später sowohl durch HTML wie auch durch JavaScript angesprochen werden. Der Fenstername darf aus Buchstaben, Ziffern sowie dem Unterstrich, jedoch nicht aus Sonderzeichen bestehen. Beachten Sie, dass nicht in jedem Fall ein Fenstername angegeben werden muss. Für den Fall, dass dessen Notation nicht gewünscht ist, wird an dessen Stelle eine leere Zeichenkette übergeben.

Die folgende Tabelle beschreibt die gängigsten Angaben, die bezüglich des Erscheinungsbildes des neu zu öffnenden Fensters gemacht werden können. Beachten Sie, dass deren Einsatz optional ist und mehrere Angaben stets durch Kommata voneinander zu trennen sind.

Anweisung	Bedeutung	Browser
`dependent`	Hierdurch wird festgelegt, ob das Elternfenster der aktuellen Fensterinstanz geschlossen werden soll. Erlaubte Werte sind hierbei `yes` für schließen und `no` für nicht schließen.	Netscape Navigator 4.0
`height`	Bestimmt die Höhe des Anzeigenfensters in Pixel.	Netscape Navigator 2.0 Internet Explorer 3.0
`hotkeys`	Legt fest, ob innerhalb der Browserinstanz die Tastaturbefehle erhalten bleiben. Erlaubte Wert hierbei sind `yes` für Tastaturbefehle und `no` für keine Tastaturbefehle.	Netscape Navigator 4.0
`innerHeight`	Definiert den vertikal zur Verfügung stehenden Anzeigebereich des neuen Fensters in Pixel.	Netscape Navigator
`innerWidth`	Definiert den horizontal zur Verfügung stehenden Anzeigebereich des neuen Fensters.	Netscape Navigator
`locationbar`	Bestimmt, ob innerhalb des aktuellen Fensters eine Adressleiste angezeigt wird. Eingesetzt werden können hierbei die Werte `yes` für eine und `no` für keine Adressleiste.	Netscape Navigator Internet Explorer 3.0
`menubar`	Legt fest, ob in dem Fenster eine Menüleiste angezeigt werden soll. Als einsetzbare Werte sind hierbei `yes` für eine und `no` für keine Menüleiste zulässig.	Netscape Navigator Internet Explorer 3.0

Tabelle 5.1: Wichtige Angaben zur Gestaltung von Fenstern

Anweisung	Bedeutung	Browser
resizeable	Bestimmt ob das Fenster in seiner Größe von dem Anwender verändert werden kann. Mögliche Werte sind hierbei **yes** für eine und **no** für keine variable Fenstergröße.	Netscape Navigator Internet Explorer 3.0
screenX	Definiert den horizontalen Abstand des Browserfensters zur linken oberen Ecke des Anzeigebereichs in Pixel.	Netscape Navigator
screenY	Definiert den vertikalen Abstand des Browserfensters zur linken oberen Ecke des Anzeigebereichs in Pixel.	Netscape Navigator
status	Legt fest, ob innerhalb der neuen Browserinstanz eine Statuszeile angezeigt werden soll. Erlaubte Werte sind hierbei **yes** für eine und **no** für keine Statuszeile.	Netscape Navigator Internet Explorer 3.0
width	Bestimmt die Breite des Anzeigefensters in Pixel	Netscape Navigator Internet Explorer 3.0

Tabelle 5.1: Wichtige Angaben zur Gestaltung von Fenstern (Forts.)

Die Tabelle zeigte deutlich, welcher enorme Handlungsspielraum im Zusammenhang mit der **open()**-Eigenschaft ermöglicht wird. Dennoch sollte, um mögliche Anzeigeprobleme umgehen zu können, auf die korrekte Browserinterpretation geachtet werden. Das nachstehende Beispiel beschreibt eine simple **open()**-Anwendung, in die einige Angaben zum Erscheinungsbild des neu zu öffnenden Fensters integriert wurden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Fenster()
{
var hinweis = window.open("hinweis.htm", "offen", "width=400,height=300,
locationbar=no")
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Fenster()">Hilfe zu dieser Seite</A>
</BODY>
</HTML>
```

Listing 5.3: Die Datei `hinweis.htm` wird in einem neuen Fenster geöffnet.

Im Zusammenhang mit der **open()**-Methode ist neben dem Aussehen des zu öffnenden Fenstes der mögliche Zugriff auf das neue Browserfenster zu beachten. Der folgende Quelltextauszug beschreibt einen Hyperlink, dessen Verweisziel `links.htm` innerhalb des neu geöffneten Fensters angezeigt werden soll. Zu diesem Zweck ist es notwendig, dass dem über **open()** geöffneten Fenster ein Name, in dem gezeigten Beispiel ist dies `offen`, zugewiesen wird. Ein möglicher Verweis auf dieses Fenster stellt sich folgendermaßen dar:

```
<A href="links.htm" target="offen">verwandte Links</A>
```

Beachten Sie, dass im Zusammenhang mit der Vergabe von Fensternamen einige Konventionen zu beachten sind. So dürfen weder Leer- und Sonderzeichen noch deutsche Umlaute verwendet werden. Zudem sind die in HML spezifizierten Fensternamen **_blank**, **_parent**, **_self** und **_top** nur dann zu verwenden, wenn dies dem gewünschten Verwendungszweck entspricht.

Abbildung 5.1: Ein neues Fenster mit definierten Eigenschaften

5.3 Fenster schließen

keine Parameter

Durch die **close()**-Methode kann ein Browserfenster geschlossen werden. **Close()** erwartet keine zusätzlichen Parameter. Eingesetzt werden kann diese Methode auf vielfältige Weise. So kann beispielsweise innerhalb eines Fensters durch den Einsatz von **close()** ein Verweis spezifiziert werden, mit dessen Hilfe sich dieses Browserfenster wieder schließen lässt. Zwar sind solche Anwendungen sinnvoll und in der Praxis weit verbreitet, es existieren aber auch andere Einsatzmöglichkeiten, mit denen sich das wahre Potenzial der **close()**-Methode ausschöpfen lässt. So kann beispielsweise aus einer HTML-Datei heraus ein Fenster geöffnet und wieder geschlossen werden. Einen solchen Fall beschreibt

das folgende Beispiel. Hier wird beim Einlesen der aufgeführten Datei ein Zweitfenster geöffnet. Dieses kann vom Hauptfenster aus, durch Anklicken des integrierten Hyperlinks, geschlossen werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Hinweis = window.open("hinweis.htm", "offen", "width=400,height=300")
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Hinweis.close()">Fenster schließen</A>
</BODY>
</HTML>
```

Listing 5.4: Das Fenster kann durch Anklicken des Hyperlinks geschlossen werden.

Im Zusammenhang mit mehreren sich neu öffnenden Fenstern gewinnt die **close()**-Methode weiter an Bedeutung. Dies gilt vor allem dann, wenn diese neuen Fenster von unterschiedlicher Größe sind. Wurde ein Fenster über JavaScript geöffnet und wird ein anderes ebenfalls über ein JavaScript aufgerufen, erfolgt dessen Anzeige innerhalb der vom ersten Fenster verwendeten Browserinstanz. Und hierbei spielt es keine Rolle, welche Größe dem zweiten Fenster zugedacht war. Vielmehr erfolgt dessen Anzeige stets innerhalb des ersten Fensters und verwendet somit auch dessen Größe. In einem solchen Fall ist es demnach ratsam, zunächst die Existenz von bereits geöffneten Fenstern abzufragen. Existiert bereits ein Fenster, ist dieses über die **close()**-Methode zu schließen und erst dann eine neue Browserinstanz zu öffnen.

5.4 Meldungsfenster

geeignet für die Fehleranalyse

Häufig verwendet und im Zusammenhang mit JavaScript auf vielerlei Art und Weise einsetzbar, ist die Methode **alert()**. Durch deren Einsatz können innerhalb einer Dialogbox textbasierte Informationen ausgegeben werden. Bei diesen Informationen kann es sich sowohl um fest kodierte Inhalte als auch um den Inhalt von Berechnungen oder Variablenwerte handeln. Für die Verwendung der **alert()**-Methode erwartet diese als Parameter lediglich eine Zeichenkette. Sinnvoll ist der Einsatz von **alert()** jedoch nicht nur im Zusammenhang mit reiner Informationsausgabe, sondern diese Methode eignet sich vorzüglich für Fehleranalysen von JavaScripts. So kann beispielsweise der korrekte Ablauf eines Programms anhand der Ausgabe der jeweiligen Variablen oder Berechnungen innerhalb eines **alert()**-Fensters ausgegeben und somit die

Funktionalität überprüft werden. Das folgende Beispiel beschreibt eine Anwendung, mit deren Hilfe dem Nutzer innerhalb eines **alert()**-Fensters angezeigt wird, wie viel Platz des ihm zur Verfügung stehenden Anzeigebereichs nicht genutzt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
var unnoetig = Math.round(1000-(screen.availHeight*screen.availWidth)/
(screen.height*screen.width)*1000)/10;
if
(unnoetig>1)
 alert("Sie vergeuden " + unnoetig + "% der Bildschirmanzeige!");
</SCRIPT>
</HEAD>
<BODY>
</HTML>
```

Listing 5.5: Ein Meldungsfenster gibt den verschwendeten Anzeigebereich aus.

Der nicht verwendete Anzeigebereich wird durch die gezeigte Funktion berechnet und dem Nutzer mittels eines **alert()**-Fensters in einem prozentualen Wert mitgeteilt. Neben dieser eigentlichen Anwendung soll das gezeigte Beispiel vor allem Demonstrationszwecken bezüglich variabler Inhalte und deren Darstellung und Ausgabe mittels zusammengesetzter Zeichenketten dienen. Auf die Integration und die spezielle Syntax von zusammengesetzten Zeichenketten wurde zwar bereits mehrfach hingewiesen, dennoch soll diese Möglichkeit hier nochmals explizit erwähnt werden. Spielt doch diese Schreibweise im Zusammenhang mit **alert()**-Fenstern eine entscheidende Rolle. Sie ermöglicht vielmehr die gleichzeitige Anzeige von mehreren Variablen innerhalb eines solchen Fensters.

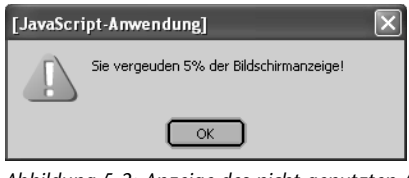


Abbildung 5.2: Anzeige des nicht genutzten Anzeigebereichs in einem Meldungsfenster

5.5 Bestätigungen

OK- und Abbrechen-Button

Um den Anwender vor Entscheidungen zu stellen und je nach dessen Auswahl zu reagieren, bietet sich der Einsatz der **confirm()**-Methode an. Hierbei handelt es sich um ein Dialogfenster, in dem die beiden Buttons OK und Abbrechen dargestellt sind. Sinnvolle Verwendungen der **con-**

firm()-Methode finden sich vor allem im Hinblick auf eine gezielte Nutzersteuerung. So könnte beispielsweise vor dem Betreten einer bestimmten Seite nochmals mittels Dialogfeld abgefragt werden, ob dies auch tatsächlich gewünscht ist. Einem solchen Einsatzgebiet der **confirm()**-Methode widmet sich das folgende Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
   function Eingang()
   {
      var Eingabe;
      Eingabe=confirm ("Wollen Sie wirklich?");
      if (Eingabe==true)
      {
      window.open("start.htm");
      }
      else
      {
      document.write("Da entgeht Ihnen aber was!!!!!!");
      }
   }
//-->
</SCRIPT>
</HEAD>
<BODY onload="Eingang()">
</BODY>
</HTML>
```

Listing 5.6: Beim Anklicken des OK-Buttons wird die Datei `start.htm` angezeigt.

Die **confirm()**-Methode erwartet als Parameter einen Text, der die Bestimmung des Dialogfelds beschreibt. Klickt der Anwender in diesem Beispiel auf den OK-Button, ist dieser Wert also auf **true** gesetzt, wird die Seite `start.htm` aufgerufen. Anderenfalls, nämlich dann, wenn der Abbrechen-Button angeklickt wird, wird ein alternativer Text, der über die **write()**-Methode erzeugt wird, dem Anwender angezeigt.

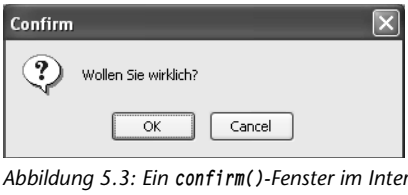


Abbildung 5.3: Ein `confirm()`-Fenster im Internet Explorer

5.6 Benutzereingaben

für Passwortabfragen

Die **prompt()**-Methode ist vielleicht eine der bekanntesten JavaScript-Anwendungen und wird vor allem im Zusammenhang mit Passwortabfragen und der Personalisierung von Internetseiten eingesetzt. Eine ebensolche Anwendung wird im Laufe dieses Abschnitts noch vorgestellt. Der vom Anwender innerhalb des **prompt()**-Fensters eingegebene Text wird von der **prompt()**-Methode zurückgeliefert und kann durch ein JavaScript weiterverarbeitet werden. Beim Einsatz der **prompt()**-Methode werden die beiden folgenden Parameter erwartet:

mögliche Parameter

- Ein vorgegebener Text, durch welchen dem Anwender die Bedeutung des Eingabefensters verdeutlicht werden sollte.
- Die Vorbelegung des Textfeldes. Hier kann ein Text oder eine leere Zeichenkette übergeben werden.

Wie bereits erwähnt, lässt sich der Einsatz der **prompt()**-Methode vorzüglich für die Personalisierung von Internetseiten verwenden. So wird durch die hier gezeigte Syntax beim Laden der Seite ein Dialogfenster geöffnet, welches den Anwender durch einen entsprechend hinweisenden Text auffordert, seinen Namen einzugeben. Zusätzlich hierzu wird innerhalb der eingaberelevanten Zeile der Text `ja hier` angezeigt. Der vom Anwender eingegebene Name wird innerhalb der `Name-Variablen` gespeichert. Dieser Wert wird anschließend durch die **write()**-Methode dazu benutzt, den Anwender persönlich zu begrüßen.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<I>Ein bisschen pers&ouml;nlicher</I>
<SCRIPT type="text/JavaScript">
<!--
var Name = prompt("Geben Sie hier \nIhren Namen ein:", "ja  hier")
if (Name)   document.write("Hallo " + Name)
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 5.7: Der eingegebene Name wird dynamisch in das Dokument geschrieben.

Beachten Sie die gesonderte Notation bei der Verwendung beider Parameter. Um beide Parameter zu verwenden, geben Sie zunächst die Zeichenkette an, die als beschreibender Text vom Anwender verstanden werden soll. Diese Zeichenkette ist fester Bestandteil des Dialogfeldes und kann vom Anwender nicht editiert werden. Der zweite Parameter,

dessen definierter Inhalt vom Anwender verändert werden kann, folgt dem ersten Parameter und wird ebenfalls in Anführungszeichen gesetzt notiert. Die beiden Parameter werden durch ein Komma voneinander getrennt.

5.7 Vor- und Zurück-Button

vergleichbar mit den bekannten Browser-Button

Bei **history** handelt es sich um ein Objekt, welches sich innerhalb der JavaScript-Objekthierarchie unter dem **window**-Objekt befindet. Dieses Objekt ermöglicht es dem Programmierer, eine ähnliche Nutzerführung umzusetzen, wie sie ein WWW-Browser bereitstellt. Bei der History-Liste eines Browserfensters handelt es sich im Prinzip um nichts anderes als eine Auflistung der zuvor besuchten Seiten. Auf die History-Liste besteht keinerlei Schreibzugriff und ein nur eingeschränkter Lesezugriff. Die Gründe für dieses Verhalten liegen auf der Hand. Ein Schreibzugriff wäre vor allem im Hinblick auf das hieraus resultierende Überschreiben der History fatal. Eine ähnlich negative Wirkung würde sich aber auch durch einen unbegrenzten Lesezugriff einstellen. Denn nicht genug damit, dass das gesamte Surf-Verhalten des Anwenders bekannt gemacht werden könnte, zusätzlich hierzu ließen sich auch die häufig an URLs angehängten Passwörter ablesen. Was freilich zu einem nicht kontrollierbaren Sicherheitsrisiko führen würde. Der grundsätzliche Zugriff auf das **history**-Objekt stellt sich folgendermaßen dar:

```
history.Methode()
```

Diese Schreibweise bezieht sich lediglich auf den Fall, dass sich der gewünschte Zugriff innerhalb der aktuellen Seite, und nicht etwa auf eine andere Seite innerhalb eines Framesets, bezieht. In einem solchen Fall müsste zusätzlich der exakte Fenstername der gezeigten Anweisung vorangestellt werden.

Einschränkungen

Eines der am häufigsten innerhalb der Navigationsleiste von WWW-Browsern benutzten Elemente ist das Zurückspringen auf die zuvor besuchte Seite. Bei einer solchen Anwendung ist zu beachten, dass nicht in jedem Fall direkt auf die vorherige Seite gesprungen wird. Hat der Nutzer beispielsweise einen Verweisanker, dessen Ziel sich innerhalb des aktuellen Fensters befindet, angeklickt, wird er zunächst wieder auf diesen geleitet. Diese Einschränkung ist demzufolge auch während der Programmierung des entsprechenden JavaScripts zu beachten. Um auch eine Seite unter den zuvor erwähnten Einschränkungen zurückspringen zu können, muss dem **history**-Objekt die Methode **back()** zugewiesen werden. Folgendes Beispiel beschreibt eine so geartete Anwendung:

```
<A href="javascript:history.back()">eine Seite zur&uuml;ck</A>
```

Die **back()**-Methode erwartet keine weiteren Parameter und wird demnach exakt in der Syntax eingesetzt, wie dies hier exemplarisch beschrieben wurde.

Die **history**-Methode **forward()** kann nur dann zum Einsatz kommen, wenn der Anwender zuvor den Zurück-Button des Browsers angeklickt hat. Das Notieren der **forward()**-Methode bewirkt, dass der Nutzer auf die Seite zurückgeleitet wird, die vor der aktuellen Seite aktiv gewesen ist. Der Einsatz einer solchen Syntax zieht demnach das gleiche Ergebnis nach sich wie der Vorwärts-Button des Browsers.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:history.forward()">
eine Seite vorw&auml;rts
</A>
</BODY>
</HTML>
```

Listing 5.8: Die nächste Seite wird angezeigt.

In dem gezeigten Beispiel wird ein Verweis deklariert. Dessen Anklicken bewirkt, dass der Anwender auf die nächste Seite, sofern zuvor ein Zurück-Vorgang ausgelöst wurde, geleitet wird.

5.8 Fenster um eine bestimmte Anzahl von Pixeln verschieben

Eltern-Fenster nicht überlagern

Im Umgang mit mehreren Fenstern und deren Verwaltung kann es immer dann zu Problemen kommen, wenn neu geöffnete Fenster an für den Anwender ungünstigen Positionen angezeigt werden. Dieser Effekt tritt zumeist dann auf, wenn neu geöffnete Fenster so geöffnet werden, dass der Inhalt des Eltern-Fensters überlagert wird und somit die hierin enthaltenen Informationen nicht mehr oder nur schwerlich lesbar sind. Dem Anwender, ohnehin zumeist nicht sonderlich erbaut von dem permanenten Öffnen neuer Browserinstanzen, bleibt hier zumeist das mühselige manuelle Verschieben oder aber das Schließen des neuen Fensters. Beide Varianten bergen Nachteile in sich, durch die die Akzeptanz eines Internetauftritts geschmälert werden kann. Um diesem Phänomen entgegenwirken zu können, empfiehlt es sich dem Anwender eine Möglichkeit zu bieten, mit deren Hilfe sich neu geöffnete Browserinstanzen um eine exakt festgelegte Pixelanzahl verschieben lassen. Die

hierfür einzusetzende Methode lautet **moveBy()** und erwartet die beiden folgenden Parameter:

erwartete Parameter

- x-Wert – Bestimmt, um wie viele Pixel das aktuelle Fenster nach rechts bzw. nach links verschoben werden soll, wobei positive Werte ein Verschieben nach rechts und negative Werte ein Verschieben nach links zur Folge haben.
- y-Wert – Bestimmt, um wie viele Pixel das aktuelle Fenster nach oben bzw. nach unten verschoben werden soll, wobei positive Werte ein Verschieben nach oben, und negative Werte ein Verschieben nach unten zur Folge haben.

Das folgende Beispiel beschreibt eine typische **moveBy()**-Anwendung. Beim Aufrufen der Datei wird gleichzeitig ein neues Browserfenster geöffnet. Dessen Startposition ist exakt die linke obere Ecke des Anzeigebereichs. Der Anwender hat nach dem Öffnen des Zweitfensters die Möglichkeit, dieses durch Anklicken des integrierten Hyperlinks zu verschieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
var Hinweis = window.open("hinweis.htm", "offen",
"width=400,height=300,left=0,top=0");
function Schiebung()
{
Hinweis.moveBy(300,100);
Hinweis.focus();
}
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Schiebung()">Fenster verschieben</A>
</BODY>
</HTML>
```

Listing 5.9: Das Fenster wird verschoben.

Aus Sicherheitsgründen ist es nicht möglich, das Browserfenster so weit zu verschieben, dass es außerhalb des sichtbaren Bildschirmbereichs angezeigt wird. Befindet sich das neu geöffnete Fenster also an einer Position, die ein weiteres Verschieben nicht möglich macht, werden die **moveBy()**-Anweisungen ignoriert und die Funktion wird nicht ordnungsgemäß ausgeführt.

5.9 Fenster an eine bestimmte Position verschieben

Im Unterschied zu der im vorangegangenen Abschnitt vorgestellten **moveBy()**-Methode wird durch **moveTo()** das Fenster nicht um eine bestimmte Pixelanzahl verschoben, sondern an einer exakt festgelegten Position angezeigt. Die **moveTo()**-Methode erwartet die beiden folgenden Parameter:

erwartete Parameter

- x- Wert – Definiert für das Browserfenster die neue Startposition von oben.
- y-Wert – Definiert für das Browserfenster die neue Startposition von links.

Das pixelgenaue Positionieren von Browserinstanzen ist vor allem dann sinnvoll, wenn ästhetische Gesichtspunkte eine tragende Rolle spielen. Ebenso wie optische Vorteile bietet die **moveTo()**-Methode aber auch die Möglichkeit, eine anwenderfreundlichere Internetseite zu kreieren. So wird es durch den Einsatz dieser Methode möglich, sich öffnende Popup-Fenster durch Mausklick vom Anwender an eine bestimmte Position verschieben zu lassen, und das ohne das Schließen dieses Fensters. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
var Hinweis = window.open("hinweis.htm", "offen",
"width=400,height=300,left=600,top=0");
function Fensterschub()
{
Hinweis.moveTo(0,0);
Hinweis.focus();
}
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Fensterschub()">Fenster verschieben</A>
</BODY>
</HTML>
```

Listing 5.10: Das Fenster wird verschoben.

Die gezeigte Syntax hat zur Folge, dass mit dem Aufrufen der Datei ein zweites Fenster mit der Startposition `600` Pixel von links sowie `0` Pixel von oben geöffnet wird. Wird der innerhalb der Datei integrierte Verweis angeklickt, ruft dies die Funktion `Fensterschub()` auf. Mittels dieser wird das geöffnete Fenster an die neue Startposition, die nun `0` Pixel

vom oberen sowie 0 Pixel vom linken Rand beträgt, verschoben. Um eine benutzerfreundliche Anwendung zu verwirklichen, sollte dem neuen Fenster die **focus()**-Methode zugewiesen werden. Durch deren Einsatz wird das neue Fenster auch nach dem Anklicken des Verweises im Vordergrund gehalten.

5.10 Fenster automatisch scrollen

nur im Netscape Navigator verfügbar

Äußerst interessante Effekte lassen sich durch den Einsatz der **srollBy()**-Methode erzielen. Durch deren Einsatz besteht die Möglichkeit, den Fensterinhalt automatisch nach oben oder unten bzw. nach links oder rechts zu scrollen. Leider steht dieses Feature bislang lediglich Netscape-Nutzern zur Verfügung. Dennoch bietet **scrollBy()** phantastische Anwendungsmöglichkeiten, um in der Praxis sinnvoll eingesetzt werden zu können.

erwartete Parameter

- x-Wert – Bestimmt, um wie viele Pixel innerhalb des Fensters nach rechts bzw. links gescrollt werden soll. Hierbei haben positive Werte ein Scrollen nach rechts und negative Wert ein Scrollen nach links zur Folge.
- x-Wert – Bestimmt, um wie viele Pixel innerhalb des Fensters nach oben bzw. unten gescrollt werden soll. Hierbei haben positive Werte ein Scrollen nach unten und negative Wert ein Scrollen nach oben zur Folge.

Eingesetzt werden kann die **scrollBy()**-Methode zwar auf vielfältige Art und Weise, vermehrt findet sie jedoch im Hinblick auf eine vereinfachtere Nutzerführung innerhalb komplexer Internetseiten Verwendung. So auch innerhalb des folgenden Beispiels. Hier wird zunächst über die **write()**-Methode innerhalb des aktuellen Dokuments die HTML-Anweisung zur Generierung von Zeilenumbrüchen **
** 50-Mal notiert. Diese Vorgehensweise dient hier jedoch lediglich einem Vorführeffekt. Im Normalfall fällt diese Vorgehensweise freilich weg, da an Stelle der hier eingefügten Zeilenumbrüche normaler Seiteninhalt angezeigt werden würde.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
for(i=1; i<=50; i++)
document.write("<BR> ");
function Anweisung(Inhalt) { return "<"+Inhalt+">" }
```

```
document.write(Anweisung("a
href='javascript:window.scrollBy(0,-550)'")+
  "noch weiter"+Anweisung("/A"))
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 5.11: Der Fensterinhalt wird automatisch gescrollt.

Durch Anklicken des dynamisch erzeugten Hyperlinks wird die Datei um weitere `550` Pixel nach unten gescrollt. Die Position vom linken Fensterrand bleibt unverändert bei einem Abstand von `0` Pixel.

5.11 Fenster automatisch scrollen (Variante 2)

Die **`scrollTo()`**-Methode ermöglicht das Scrollen der Seite zu einer exakt festgelegten Position innerhalb des Dokuments. Nicht nur auf den ersten Blick handelt es sich hierbei um eine zu **`scrollBy()`** identische Methode. Nur wird bei **`scrollTo()`** an eine bestimmte Position gescrollt, während durch **`scrollBy()`** um eine festgelegte Anzahl von Pixel gescrollt wird. Genau wie **`scrollBy()`** erwartet auch die **`scrollTo()`**-Methode zwei Parameter. Deren Bedeutung stellt sich folgendermaßen dar:

erwartete Parameter

- x-Wert von links – Bestimmt die Position des Fensters von der linken oberen Ecke aus betrachtet.
- y-Wert von oben – Bestimmt die Position des Fensters von der linken oberen Ecke aus betrachtet.

Das den Einsatz der **`scrollTo()`**-Methode veranschaulichende Beispiel zeigt, wie sich eine Seite mit geringem Aufwand elegant durch das Anklicken eines Hyperlinks an einen exakt definierten Punkt der Seite scrollen lässt. Dieses Scrollen findet hierbei jedoch tatsächlich als Scrollen und nicht, wie bei der Definition eines herkömmlichen Hyperlinks, als Sprung statt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function scrollen()
{
x=y=1;
self.scrollTo(x,y);
}
function rollen()
```

```
{
setTimeout("weiter()", 1);}
function weiter()
{
self.scrollTo(x,y);
if (y<7250)
{
rollen();
y+=3;
}
}
// -->
</SCRIPT>
<BODY onload="scrollen()">
<A href="javascript:rollen()">find mich!</A>
<BR><BR><BR><BR><BR><BR><BR><BR>
<BR><BR>Hier bin ich<BR><BR><BR>
</BODY>
</HTML>
```

Listing 5.12: Das Fenster wird bis zu dem Text gescrollt.

Innerhalb der Syntax wird die Funktion `rollen()` aufgerufen. In dieser Funktion wird die Methode **`setTimeout()`** verwendet. Als Parameter werden hier der Funktionsname `weiter()` sowie `1` übergeben.

5.12 Drucken

keine Parameter

Einfach in der Anwendung, groß in der Wirkung. So ließe sich die **`print()`**-Methode am ehesten beschreiben. Durch deren Einsatz lässt sich der Inhalt einer WWW-Seite ausdrucken. Es handelt sich hierbei um das gleiche Prinzip, welches auch in WWW-Browsern als Menüpunkt Anwendung findet. Sinnvoll erscheint der Einsatz dieser Methode vor allem im Zusammenhang mit informationsbasierten Internetseiten. Aber auch im Hinblick auf Formulare für Bestellsysteme im Internet oder ähnliche Anwendungen ist dies ein guter Weg, um mit geringem Aufwand ein unvergleichliches Mehr an Nutzerfreundlichkeit realisieren zu können. Die **`print()`**-Methode erwartet keine Parameter. Der Einsatz stellt sich auf die folgende unkomplizierte Art dar:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:window.print()">
<img src="print.gif" border="0">
```

```
</A>
</BODY>
</HTML>
```

Listing 5.13: Das Anklicken der Grafik öffnet den Druckdialog.

Durch Anklicken der innerhalb des Beispiels integrierten Grafik wird der WWW-Browser dazu veranlasst, das Dialogfeld „Drucken“ anzuzeigen. Es findet demnach also kein direktes und automatisches Ausdrucken der Seite statt. Vielmehr kann der Anwender nochmals eigenständig entscheiden, ob der Druckvorgang tatsächlich fortgesetzt werden soll.

5.13 Statuszeile verwenden

Ist eine Statuszeile vorhanden?

Es wurde bereits auf die Vor- und Nachteile des Eingreifens in die Statuszeile eingegangen. Dies soll nun freilich an dieser Stelle nicht wiederholt werden. Gleichwohl ist es im Zusammenhang mit einer entsprechenden Textbelegung der Statuszeile, wenn auch nicht notwendig, so doch in vielen Fällen sinnvoll, zu erfahren, ob innerhalb der aktuellen Browserinstanz überhaupt eine Statuszeile angezeigt wird. Möglich ist eine solche Abfrage anhand der **statusbar**-Eigenschaft, welche die beiden folgenden Zustände haben kann:

mögliche Werte

- **true** – Die Statuszeile ist sichtbar.
- **false** – Die Statuszeile ist nicht sichtbar.

Auch im Zusammenhang mit der Eigenschaft **statusbar** sind die möglichen Anwendungen zahlreich. Exemplarisch soll für deren Einsatz die folgende Syntax zeigen, in welcher Form sich praktische Anwendungen realisieren lassen. Innerhalb des folgenden Beispiels wird ein Formular mit dem Namen `Formular` definiert. In diesem befinden sich ein einzeiliges Eingabefeld sowie ein Formular-Button. Der Anwender kann einen beliebigen Text innerhalb des Eingabefeldes notieren, welcher nach dem Anklicken des Buttons in der Statuszeile angezeigt wird. Diese Vorgehensweise kann jedoch nur so lange aufrechterhalten werden, wie innerhalb des Browsers auch tatsächlich eine Statuszeile angezeigt wird. Um deren Existenz abzufragen, wurde die Funktion `Statuszeile()` für diesen Verwendungszweck optimiert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Statuszeile(Inhalt)
{
if(statusbar.visible == true) window.defaultStatus = Inhalt;
```

```
else alert(Inhalt);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" size="12" name="Eingabe">
<INPUT type="button" value="Statuszeile" onclick="Statuszeile(document.Formu-
lar.Eingabe.value)";
</FORM>
</BODY>
</HTML>
```

Listing 5.14: Besitzt das Fenster eine Statuszeile?

Die Funktion `Statuszeile()` überprüft zunächst, ob die aktuelle Browserinstanz eine Statuszeile besitzt. Ist das der Fall, also wird der Wert **`true`** zurückgeliefert, erscheint der innerhalb des Eingabefeldes notierte Text in dieser Statuszeile. Um dem Anwender aber auch im Falle des Nichtvorhandenseins der Statuszeile seine von ihm eingegebenen Informationen nicht vorenthalten zu müssen, werden diese bei der Rücklieferung des **`false`**-Werts in einem **`alert()`**-Fenster angezeigt.

Abbildung 5.4: Der Text des Eingabefeldes wird in der Statuszeile angezeigt.

5.14 Inhalt der Statuszeile

Die Eigenschaft **`defaultStatus`** speichert den Inhalt der Statuszeile. Dieser definierte Inhalt wird so lange angezeigt, bis ein anderes Ereignis, zum Beispiel das Überfahren eines Verweises mit der Maus, eintritt. Zweifelhaft ist der Zugriff auf die Statuszeile trotz einiger möglicher optisch reizvoller Aspekte aber in jedem Fall. Im nachfolgenden Beispiel wird mit dem Laden der Seite in der Statuszeile der Browserinstanz ein Begrüßungstext angezeigt. Dieser bleibt dem Anwender so lange erhalten, bis ein anderes, die Statuszeile betreffendes, Ereignis eintritt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
window.defaultStatus = "Herzlich willkommen";
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 5.15: Der Text wird in der Statuszeile angezeigt.

Anhand der aufgeführten Syntax wird innerhalb der Statuszeile des Browsers eine Nachricht angezeigt. Dieser Zustand bleibt so lange erhalten, bis ein Ereignis, also beispielsweise das Überfahren eines Hyperlinks mit der Maus, eintritt.

5.15 Zwei Fenster gleichzeitig laden

Die **`open()`**-Methode ermöglicht das Öffnen einer neuen Browserinstanz mit exakt definierten Eigenschaften, wie beispielsweise deren Größe. In vielen Fällen wird aber nun nicht nur eine, sondern werden zwei sich gleichzeitig öffnende Browserfenster benötigt. Das folgende Script-Beispiel zeigt eine für diese Anforderung optimierte Anwendung.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function oeffneFenster()
{
window.open('template.htm', 'Fenster1',
'toolbar=no,menubar=no,width=200,height=200,left=0,top=0,')
window.open('test.htm', 'Fenster2',
'toolbar=no,menubar=no,width=200,height=200,left=0,top=200,')
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="oeffneFenster()">
</BODY>
</HTML>
```

Listing 5.16: Die beiden Fenster werden untereinander angezeigt.

Den beiden sich öffnenden Fenstern wird jeweils ein eindeutiger Bezeichner, in diesem Beispiel also `Fenster1` und `Fenster2`, zugewiesen. In

jede der beiden sich neu öffnenden Browserinstanzen wird eine andere Datei angezeigt. Durch Anpassung bzw. Erweiterung des gezeigten Scripts lässt sich zusätzlich eine Steuerung, mit deren Hilfe sich die einzelnen Fenster gezielt ansprechen lassen, realisieren. Denkbar wäre hierbei beispielsweise die Möglichkeit, beide Fenster auf Tastendruck zu schließen oder zu verschieben. Die Eigenschaften beider Fenster wie z.B. deren Größe und jeweilige Startpositionen lassen sich ebenfalls durch simple Syntaxmodifikationen anpassen.

5.16 Höhe des Anzeigebereichs

Durch die Eigenschaft **innerHeight** ist es möglich, die Höhe des Anzeigebereichs eines Browserfensters auszulesen und diese zu verändern. Angewandt werden kann eine solche Syntax allerdings lediglich im Zusammenhang mit für den Netscape Navigator ab Version 4.0 optimierten Seiten. Der Einsatz dieser Eigenschaft ist vor allem immer dann sinnvoll, wenn eine direkte Abhängigkeit zwischen der Höhe des Anzeigebereichs und der absoluten Positionierung von Elementen besteht bzw. eine solche hergestellt werden soll. Es ist vor dem Einsatz der **innerHeight**-Eigenschaft zu bedenken, dass die festgelegte Höhe des Anzeigebereichs nicht mit der Höhe des Browserfensters gleichgesetzt oder verwechselt werden darf. Denn schließlich werden durch die hier aufgeführte Eigenschaft nicht solche Elemente berücksichtigt, die zusätzlich für die tatsächliche Größe des Browserfensters verantwortlich sind. Dies sind beispielsweise die Status- und die Menüleiste.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
if (window.innerHeight < 1000)
{
alert("Bitte maximieren Ihr das Fenster.")
}
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 5.17: Ein Hinweis für den Anwender

In dem gezeigten Beispiel wird, dies lässt sich allerdings nur mit dem Netscape Navigator nachvollziehen, zunächst abgefragt, ob die Höhe des Anzeigebereichs kleiner als `1.000` Pixel ist. Für den Fall, dass dies zulässig, wird ein Meldungsfenster ausgegeben, durch welches der Anwender ermutigt wird, die Größe des Browserfensters zu maximieren.

5.17 Breite des Anzeigebereichs

Um die Breite des Anzeigebereichs eines Fensters zu speichern, steht die Methode **innerWidth** zur Verfügung. Durch deren Einsatz wird es möglich, nicht nur die aktuelle Breite auszulesen, sondern diese aktiv zu ändern und somit gezielt auf spezielle Anforderungen der darzustellenden Seite reagieren zu können. Neben dieser Möglichkeit der besseren Darstellbarkeit eignet sich die **innerWidth**-Methode jedoch auch noch für Anwendungen, die sich des absoluten Positionierens von Elementen bedienen. Das folgende Beispiel beschreibt ein, wenn sich auch dies nicht auf den ersten Blick erkennen lassen mag, für viele Anwendungen verwendbares Script, welches sich der **innerWidth**-Methode bedient. Durch das gezeigte Script wird es möglich, direkt auf die Gegegebenheiten auf der Client-Seite zu reagieren und somit die eigene Internetseite diesen optimal anzupassen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
if (window.innerWidth < 500)
window.innerWidth = 750;
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 5.18: Das Fenster wird bei Bedarf vergrößert.

Das gezeigte Script wird automatisch mit dem Aufrufen der aktuellen Seite geladen. Ist die Breite des Anzeigenbereichs auf Client-Seite kleiner als `500` Pixel, wird dieses Browserfenster automatisch auf `750` Pixel vergrößert. Von Vorteil ist dieses Script immer dann, wenn das eigene Projekt eine Minimalanforderung an die Anzeigenbreite des Browserfensters stellt, um entweder optisch oder aber auch funktionell einwandfrei darstellbar zu sein.

5.18 Vollbild-Fenster

Die Seite kann nur über die Tastatur geschlossen werden.

Zweifellos fragwürdig und für den Anwender in vielen Fällen ärgerlich ist das automatische Anpassen der Fenstergröße in einen Vollbildmodus. In diesem Abschnitt wird beschrieben, wie mit dem Laden einer Seite ein zweites Browserfenster geöffnet und hierin eine neue Seite angezeigt werden kann. Das Besondere an dieser neuen Seite ist, dass diese über die gesamte zur Verfügung stehende Anzeigefläche des Monitors

dargestellt wird. Das Schließen des Fensters durch den Anwender ist daher nur über die gängigen Tastaturkürzel innerhalb des jeweiligen Browsers möglich. Selbstverständlich bietet es sich aus diesem Grund an, über die **window**-Eigenschaft **close()** eine Möglichkeit zum Schließen durch den Anwender anzubieten.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Vollbild()
{
window.open("http://www.bosstones.com","bosstones","fullscreen=yes")
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="Vollbild()">
</BODY>
</HTML>
```

Listing 5.19: Die Seite hat jetzt sehr viel Platz.

Beim Laden der aktuellen Datei wird die Funktion `Vollbild()` mittels des Event Handlers **onload** aufgerufen. Das neue Fenster wird über die **open()**-Methode des **window**-Objekts geöffnet. Auf diese Syntax soll an dieser Stelle allerdings nicht noch mal eingegangen werden. Interessierte Leser könne die Varianten der **open()**-Methode innerhalb des Abschnitts 5.2 nachlesen. Für das hier gezeigte Beispiel ist lediglich die Anweisung **fullscreen=yes** interessant. Denn eben diese ermöglicht die Darstellung der neuen Seite in einem Vollbildmodus.

5.19 Fenster immer zentrieren

effektvolle Werbebotschaften

Zwar funktionell nicht bedeutsam, spielt die Positionierung von neu zu öffnenden Fenstern gerade in ästhetischer Hinsicht eine zunehmende Rolle. Für viele Anwender ist die Art der Positionierung von Popup-Fenstern bereits dahingehend bedeutsam, ob die Informationen, bei denen es sich wohl zumeist um Werbebotschaften handeln dürfte, tatsächlich aufgenommen werden. Das hier gezeigte JavaScript bietet zwar auch nicht die Lösung für alle im Zusammenhang mit der Aufnahmenfähigkeit des Anwenders auftretenden Probleme, es kann jedoch zu einer ästhetischen Verbesserung bzw. Optimierung von Internetseiten herangezogen werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Fenster = null;
function neuesFenster(Seite,Name,Breite,Hoehe){
linkePosition = (screen.width) ? (screen.width-Breite)/2 : 0;
oberePosition = (screen.height) ? (screen.height-Hoehe)/2 : 0;
Eigenschaften ='height='+Hoehe+',width='+Breite+',top='+ oberePosition
+',left='+ linkePosition +',scrollbars=no,resizable=no'
Fenster = window.open(Seite,Name,Eigenschaften)
if(Fenster.window.focus){Fenster.window.focus();}
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="burning.htm" onclick="neuesFen-
ster(this.href,'Name','300','400','yes');return false">zentriertes neues Fen-
ster</A>
</BODY>
</HTML>
```

Listing 5.20: Der Anwender kann das Fenster aber dennoch an eine andere Position verschieben.

Das neu zu öffnende Fenster besitzt eine fest definierte Größe von `300` mal `400` Pixel. Innerhalb der Funktion `neuesFenster()` wird die Startposition vom oberen und linken Rand über die beiden Anweisungen **`screen.width`** und **`screen.height`** ausgelesen. Um das neue Fenster beim Öffnen immer zentriert darstellen zu können, müssen zunächst die Gesamtbreite und Höhe des Anzeigenbereichs ermittelt werden. Anschließend werden diese Werte durch zwei geteilt. Die hieraus erhaltenen Werte werden innerhalb der Eigenschaftendeklaration des neuen Fensters für die Startpositionen **`left`** und **`top`** verwendet.

5.20 Animiertes öffnendes Fenster

schrittweises Vergrößern

Dieses JavaScript-Programm ermöglicht das Öffnen eines neuen Fensters. An sich ist dies nun nichts Besonderes. Der Clou an diesem Script ist jedoch, dass das neue Fenster schrittweise vergrößert wird. Geöffnet wird es zunächst am linken oberen Rand des Anzeigefensters. Anschließend wird es in Schritten von `5` Pixel und mit einer jeweiligen Verzögerung von `5` Millisekunden aufgezogen. Dieser Vorgang wiederholt sich so lange, bis das neue Fenster die gesamte Fläche des Anzeigebereichs eingenommen hat. Erst wenn dieser Vorgang abgeschlossen ist, wird die Seite *http://www.mysite.com* in dem neuen Fenster geöffnet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var URL= 'http://www.mysite.com';
var Breite=100;
var Hoehe=100;
var x=5;
function animiertesFenster()
{
neuesFenster=window.open("","","")
if (!document.layers&&!document.all)
{
win2.location=URL
return
}
neuesFenster.resizeTo(100,100)
neuesFenster.moveTo(0,0)
schrittweisesEinfuehren()
}
function schrittweisesEinfuehren()
{
if (Hoehe>=screen.availHeight)
x=0
neuesFenster.resizeBy(5,x)
Hoehe+=5
Breite+=5
if (Breite>=screen.width-5)
{
neuesFenster.location=URL
return
}
setTimeout("schrittweisesEinfuehren()",50)
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:animiertesFenster()">Fenster &ouml;ffnen</A>
</BODY>
</HTML>
```

Listing 5.21: Schrittweises Öffnen des Fensters

Zu Beginn des Programms werden der URL, die Breite und die Höhe des zu öffnenden Fensters als Variablen deklariert. Im nächsten Schritt wird über **`window.open()`** ein neues Browserfenster geöffnet. Dieses wird über

resizeTo() in einer Größe von 100 mal 100 Pixel angezeigt. Da der Prozess des Einschiebens von der linken oberen Ecke des Bildschirms beginnen soll, wird die Anweisung **moveTo(0,0)** verwendet. Die Funktion schrittweisesEinfuehren() sorgt für das Einschieben des Fensters. Hierzu wird zunächst überprüft, ob der Wert der Variablen Hoehe größer oder gleich der maximalen Anzeigehöhe des Fensters ist. Dieser Maximalwert wird über die Eigenschaft **availHeight** ermittelt. Ist diese Bedingung erfüllt, wird das Fenster um 5 Pixel nach rechts vergrößert. Hierzu dient die **resizeBy()**-Methode. Beachten Sie, dass hier der erste Wert die Größenänderung nach rechts umsetzt. Der zweite Wert, der eigentlich für eine Größenänderung nach unten zuständig ist, wird zunächst auf 0 gesetzt. Anschließend wird den beiden Variablen Hoehe und Breite ein Wert von plus 5 Pixel zugewiesen. Um die Breite des Fensters zu überprüfen, wird die **width**-Eigenschaft des **screen**-Objekts verwendet. Um die Funktion schrittweisesEinfuehren() alle 50 Millisekunden zu wiederholen, wird **setTimeout()** verwendet. Erst wenn der gesamte Anzeigebereich mit dem neuen Browserfenster ausgefüllt ist, wird die Seite über **location=**URL geladen.

5.21 Fenster ohne Rahmen öffnen

keine Titelleiste

Bei der in diesem Abschnitt vorgestellten Variante der **open()**-Methode des **window**-Objetks handelt es sich um eine fragwürdige Anwendung. Das Prinzip des in diesem Beispiel zu öffnenden Fensters ist denkbar einfach. Wird die gezeigte HTML-Datei aufgerufen, öffnet sich ein zweites Fenster. Dieser Effekt mag nun auf den ersten Blick nicht ungewöhnlich erscheinen. Das Ungewöhnliche an dieser Funktion stellt vielmehr die Anzeigevariante des neu generierten Fensters dar. Denn dieses wird ohne die bislang nicht auszublendende Titelleiste angezeigt. Erreicht wird dieses Resultat durch die Anweisung **fullscreen** innerhalb der Darstellungsdeklaration des neuen Fensters. Der Parameter **1** bzw. **yes** führt dazu, dass das neue Fenster im Vollbildmodus, also ohne die Anzeige der Titelleiste, dargestellt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function rahmenlosesFenster()
{
var Breite=250;
var Hoehe=200;
var oben=125;
var links=102;
var URL="http://www.bosstones.com";
```

```
auf = window.open("url",'x','fullscreen=1,toolbar=no,location=no,
directories=no,status=no,menubar=no,scrollbars=no');
auf.focus();
auf.blur();
auf.resizeTo(Breite,Hoehe);
auf.moveTo(oben,links);
auf.location=URL;
auf.focus();
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="rahmenlosesFenster()">
</BODY>
</HTML>
```

Listing 5.22: Das Fenster wird ohne Rahmen dargestellt.

So ästhetisch ein durch diese Syntax gestaltetes Fenster auch sein mag, problematisch ist diese Syntax allemal. Nicht nur, dass die Anweisung **fullscreen** lediglich vom Internet Explorer unterstützt wird, bringt diese Syntaxform auch im Hinblick auf die Benutzerführung Nachteile mit sich. Der Anwender besitzt beim Umgang mit einem Fullscreen-Fenster nicht die Möglichkeit, dieses über dessen Titelleiste zu schließen. Problematischer wird ein solches Fenster aber noch, wenn dessen Größe den gesamten Anzeigebereich einnimmt. Hier besitzt der Anwender nun keine Möglichkeit mehr, dass Fenster auf einfache Weise zu schließen. Demzufolge sollte im Zusammenhang mit der **fullscreen**-Anweisung stets ein Button bzw. Hyperlink angeboten werden, der über **close()** die Möglichkeit bietet, das aktuelle Fenster schnell und unproblematisch wieder schließen zu können.

5.22 MouseOver-Text – Neues Fenster

Wie sich eine Symbiose aus einem textbasierten Hyperlink, dem Event-Handler **mouseover** und der **window**-Methode **open()** bewerkstelligen lässt, wird in diesem Abschnitt vorgestellt. Die hierfür eingesetzte Syntax birgt an sich nichts Neues in sich. Innerhalb der Funktion `oeffneFenster()` wird über **window.open()** die Definition des neuen Fensters eingeleitet. Als Parameter der **open()**-Methode werden hier der URL, die Größe und Breite sowie einige fundamentale Anweisungen zur Darstellung des neuen Fensters spezifiziert. Innerhalb des Dateikörpers wird ein Hyperlink notiert. Dieser dient zwei Anwendungszwecken. Zum einen wird durch dessen Anklicken die Seite `musik.htm` geöffnet und zum anderen wird, wenn das Ereignis **onmousover** eintritt, die Funktion `oeffneFenster()` ausgelöst.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function oeffneFenster()
{
window.open("http://www.offspring.com","neu","toolbar=no,location=no,
directories=no,status=no,menubar=no,scrollbars=no,resizable=no,
width=300,height=300");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="musik.htm" onmouseover="oeffneFenster(); return true;">The Off-
spring</A>
</BODY>
</HTML>
```

Listing 5.23: Ein neues Fenster dank `onmouseover`

So sinnvoll die gezeigte Syntax in einigen Fällen auch sein mag, auch bei deren Verwendung gilt: Ein Übermaß an sich permanent öffnenden Fenstern ist für den Anwender letztendlich wenig erbaulich. Aus diesem Grund sollte die gezeigte Syntaxform nur in stark begrenztem Umfang eingesetzt werden.

5.23 Adresszeile

Anhand der **`location`**-Eigenschaft kann das Vorhandensein der Adresszeile des Browsers ausgelesen werden. Zumeist hat die Erkenntnis, ob tatsächlich eine Adresszeile sichtbar ist, zwar lediglich mit ästhetischen Gesichtspunkten zu tun, dennoch kann der Einsatz der **`location`**-Eigenschaft sinnvoll sein. Die Adresszeile selbst stellt ein Objekt dar, welches eine der folgenden Eigenschaften besitzen kann:

- **`true`** – Die Statuszeile ist sichtbar.
- **`false`** – Die Statuszeile ist nicht sichtbar.

Das folgende Beispiel beschreibt eine mögliche praktische Umsetzung der **`location`**-Eigenschaft. Beim Einlesen der Datei wird zunächst ein Zweitfenster mit dem Namen `neues` geöffnet. Dieses neue Fenster besitzt zunächst keine Adresszeile:

```
<HTML>
<HEAD>
```

```
<SCRIPT type="text/JavaScript">
neuesFenster = window.open("http://www.bosstones.com", "neues",
"width=400,height=300");
function Adresszeile()
{
if(neuesFenster.locationbar.visible ==false)
{
neuesFenster.close();
ZweitFenster = window.open("http://www.bosstones.com", "neues",
"width=400,height=300,location");
ZweitFenster.focus();
}
}
</SCRIPT>
</HEAD>
<BODY onload="Adresszeile()">
</BODY>
</HTML>
```

Listing 5.24: Ist die Adresszeile vorhanden?

Die Funktion `Adresszeile()` prüft, ob das neue Fenster eine eigene URL-Zeile besitzt. Ist dies nicht der Fall, wird dieses Fenster geschlossen und an dessen Stelle ein neues Fenster, dieses Mal allerdings mit einer URL-Zeile, geöffnet.

5.24 Menüleiste

Die **window**-Eigenschaft **menubar** ermöglicht die Abfrage, ob die aktuelle Browserinstanz eine eigene Menüleiste besitzt. Innerhalb dieser Menüleiste werden beispielsweise all die Befehle angezeigt, die für eine Bearbeitung von HTML-Seiten notwendig sein können. **Menubar** kann eine der folgenden Eigenschaften besitzen:

- **true** – die Statuszeile ist sichtbar
- **false** – die Statuszeile ist nicht sichtbar

Beachten Sie, dass der aktuelle Wert der **menubar**-Eigenschaft nur lesen, nicht aber per JavaScript-Zugriff schreiben lässt. Das Vorhandensein der Menüleiste ist vor allem für Personen interessant, denen an der Bearbeitung der aktuellen Seite gelegen ist. Ein für diese Zielgruppe optimiertes Beispiel beschreibt die folgende Syntax.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Menueleiste()
```

```
{
if(window.menubar.visible == true)
alert("Sie können die Seite editieren");
}
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Menueleiste()">bearbeiten?</A>
</BODY>
</HTML>
```

Listing 5.25: Kann die Seite bearbeitet werden?

Innerhalb der Funktion **Menueleiste()** wird zunächst über **visible == true** abgefragt, ob die aktuelle Browserinstanz eine Menüleiste besitzt. Ist dies der Fall, wird innerhalb eines **alert()**-Fensters ein entsprechender diesbezüglicher Hinweis ausgegeben.

5.25 Tool-Tipp-Fenster

eine etwas andere Anwendung

Ein Tool-Tipp-Fenster wird in herkömmlichem HTML seit Version 4.0 durch das Notieren des **title**-Tags generiert. Ein hiermit erzeugtes Tool-Tipp-Fenster kann für den Anwender nützliche Informationen zu der entsprechenden Textpassage oder einer Grafik enthalten. Zwar für den gleichen Verwendungszweck gedacht, nämlich weiterführende Informationen bekannt zu geben, jedoch per JavaScript realisiert, ist das hier vorgestellte Tool-Tipp-Fenster zu betrachten. Die gezeigte Syntax bedient sich zweier Komponenten: Zum einen des Event-Handlers **onmouseover** und zum anderen der **open()**-Methode des **window**-Objekts. Zunächst die vollständige Syntax des Beispiels:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Hilfefenster(Nachricht)
{
var Hilfe = open('', 'neu_fenster', 'width=100,height=10');
var Hinweis = Hilfe.document;
Hinweis.open();
hinweis.writeln('<HTML><HEAD>');
hinweis.writeln('<title>Hilfe zu dieser Seite</title>');
hinweis.writeln('</HEAD><BODY>');
hinweis.writeln('<B>' + Nachricht + '</B>');
hinweis.writeln('</BODY></HTML>');
}
```

```
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="login.htm">Login</A>
<A href="info.htm" onmouseover="Hilfefenster('Geben
Sie hier Ihr Passwort ein!')"><IMG src="help.gif"></A>
</BODY>
</HTML>
```

Listing 5.26: Das neue Fenster besitzt bereits einen vordefinierten Inhalt.

Innerhalb des Dateikörpers wird eine Grafik integriert, die mit dem Event-Handler **onmouseover** versehen wird. Wird also mit der Maus über die Grafik gefahren, ruft dies die Funktion `Hilfefenster()` auf. In dieser wird zunächst ein neues Fenster mit den Maßen `100` mal `10` Pixel geöffnet und dessen Inhalt dynamisch generiert. Alle für die korrekte Anzeige dieses neuen Fensters elementaren Bestandteile der HTML-Datei werden hierbei vorgegeben und sind innerhalb der Funktion fest kodiert. Als in gewissem Maß dynamisch zu betrachten ist vielmehr der Text, der innerhalb des neu erzeugten Fensters angezeigt werden soll. Dieser wird über die Anweisung `Nachricht` spezifiziert, deren Inhalt jedoch nicht direkt in der JavaScript-Funktion, sondern innerhalb der **onmouseover**-Anweisung notiert wird. Der Vorteil dieser Herangehensweise liegt, zumindest wenn mehrere Grafiken mit Tool-Tipp-Fenstern versehen werden sollen, auf der Hand: Es wird lediglich eine Funktion benötigt, wobei der Inhalt der Anweisung `Nachricht` von Grafik zu Grafik variabel gestaltet werden kann.

5.26 Statuszeilen-Text mit festgelegter Zeichenanzahl

Teilkette extrahieren

Auf den ersten und vielleicht auch den zweiten Blick erscheint die hier vorgestellte JavaScript-Anwendung wohl eher als Spielerei, aber auch diese lässt sich für ernsthafte Funktionen nutzen. Das Grundprinzip der gezeigten Syntax beruht auf der **substring()**-Methode. Diese extrahiert aus einer globalen Zeichenkette eine Teilkette. Von welcher Position an und bis zu welcher Position diese Extrahierung stattfinden soll, wird durch die Parameter der **substring()**-Methode spezifiziert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
Statustext="Beachten Sie bitte alle Hinweise!"
```

```
window.status=Statustext.substring(0,32)
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 5.27: Der Text wird nicht vollständig angezeigt.

Das gezeigte Beispiel hat zur Folge, dass bereits mit dem Laden der Seite innerhalb der Statuszeile der definierte Text angezeigt wird. Hierzu wird der Variablen `Statustext` der anzuzeigende Text zugewiesen. Durch die Angabe **`window.status`** wird der Browser veranlasst, die nachfolgenden Anweisungen innerhalb der Statuszeile anzuzeigen. Dazu gehört zunächst lediglich der Text. Zusätzlich zu diesem Standardverhalten wird in dem gezeigten Beispiel festgelegt, dass alle Zeichen der anzuzeigenden Zeichenkette dargestellt werden. Einzige Ausnahme bildet hierbei das letzte Zeichen der Zeichenkette. Erreicht wird dieser Effekt durch die Verwendung der **`substring()`**-Methode und die beiden als Parameter notierten numerischen Werte `0`, was die Anzeige vom ersten Buchstaben an bedeutet, sowie `32`, was das letzte darzustellende Zeichen spezifiziert. Da die Gesamtanzahl der innerhalb der Statuszeile anzuzeigenden Zeichen *33* ist, wird das letzte Zeichen nicht mit dargestellt.

5.27 Scrollender Statuszeilentext

Nachteile der Belegung der Statuszeile

Texte, die innerhalb der Statuszeile angezeigt werden, sind häufig ein Ärgernis. Verhindern diese doch die Anzeige notwendiger Informationen. So wird die Statuszeile vom Anwender beispielsweise häufig dazu genutzt herauszufinden, welches Ziel ein Hyperlink besitzt. Dennoch soll hier eine Möglichkeit gezeigt werden, einen Text in der Statuszeile anzuzeigen. Dieser Text ist nicht statisch, sondern wird als von rechts nach links scrollender Statuszeilentext angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Statustext="
Bitte beachten Sie alle Hinweise ... Es kommen noch mehr..."
i=0
function scrollenderText()
{
window.status=
Statustext.substring(i,Statustext.length)+Statustext.substring(0,i-1)
```

```
if(i<Statustext.length)
{i++}
else
{i=0}
setTimeout("scrollenderText()",200)
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="scrollenderText()">
</BODY>
</HTML>
```

Listing 5.28: Der Text scrollt in der Statuszeile.

Die Funktion `scrollenderText()` wird durch den Einsatz des Event-Handlers **`onload`** mit dem Laden der Seite ausgeführt. Innerhalb des Script-Bereichs wird zunächst die Variable `Statustext` mit dem anzuzeigenden Text deklariert. Durch die Eigenschaft **`status`** wird auf die Statuszeile des Browsers zugegriffen. Als Wert werden dieser Eigenschaft die Variable `Statustext` sowie die **`substring()`**-Methode zugewiesen. Diese Methode extrahiert aus einer Zeichenkette eine Teilzeichenkette. Als Parameter werden die Position des ersten zu extrahierenden Zeichens sowie die Position des ersten nicht mehr zu extrahierenden Zeichens angegeben. Die Parameterwerte sind die Variable **`i`**, welcher der Wert `0` zugewiesen wurde, sowie die Gesamtzahl der Zeichen des Variablenwertes von `Statustext`. Dies bedeutet, dass alle Zeichen der Zeichenkette angezeigt werden. Durch die Anweisung **`substring`**`(0,i-1)` wird nun bestimmt, dass alle Zeichen bis auf das letzte in der Statuszeile angezeigt werden. Die **`if`**-Abfrage überprüft, ob der Wert von **`i`** kleiner als die Gesamtanzahl der Zeichen der Zeichenkette ist. Ist dies der Fall, wird der Wert von **`i`** um eins erhöht. Ist diese Bedingung nicht mehr erfüllt, wird der Wert der Variablen **`i`** wieder auf `0` gesetzt. Abschließend wird über die **`setTimeout()`**-Methode festgelegt, dass die Funktion **`scrollenderText()`** alle `200` Millisekunden neu ausgeführt wird.

5.28 Neue Seite durch Button-Klick

Textbasierte Hyperlinks sind ein bekanntes Stilmittel, um Verweise auf andere Dokumente zu realisieren. Ebenso ist es aber auch möglich, Formular-Buttons zum Verweisen zu verwenden. Ein solches Beispiel beschreibt die folgende Syntax. Hier wird innerhalb der HTML-Datei ein Formular-Button angezeigt. Wird dieser angeklickt, erfolgt der Aufruf der Seite *http://www.specials.com*.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Weiterleitung()
{
location.href= "http://www.specials.com";
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT type="button" onclick="Weiterleitung()" value="Klick mich!">
</FORM>
</BODY>
</HTML>
```

Listing 5.29: Der Aufruf einer neuen Seite

Die Funktion `Weiterleitung()` wird durch Anklicken des Formular-Buttons ausgelöst. Hierin wird über die **`href`**-Eigenschaft des **`location`**-Objekts ein neuer URI aufgerufen. Die aufzurufende Datei wird der **`href`**-Eigenschaft als Wert zugewiesen.

5.29 Onmouseover Alert

Das hier gezeigte JavaScript eignet sich vorzüglich, um Anwender in tiefe Verzweiflung zu stürzen. Aus diesem Grund sollte gut überlegt werden, an welcher Stelle dieses Script eingesetzt wird. Dem Anwender wird beim Laden der Seite ein Hyperlink angezeigt. Versucht er, diesen anzuklicken, öffnet sich ein Meldungsfenster mit dem Hinweis `Das funktioniert so nicht!`

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function ScherzLink()
{
 alert("Das funktioniert so nicht!");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
```

```
<FORM>
<A href="#" Onmouseover="ScherzLink()">Klick mich!</A>
</FORM>
</BODY>
</HTML>
```

Listing 5.30: Ein bisschen anstrengend ist diese Funktion schon ...

Die Funktion `ScherzLink()` wird durch den Event-Handler **`onmouseover`** ausgelöst. Innerhalb dieser Funktion wird über die **`alert()`**-Methode ein Meldungsfenster generiert. Als Information wird hierin der Text `Das funktioniert so nicht!` angezeigt.

5.30 Mehrere Grafiken öffnen unterschiedliche Fenster

Parameter verwenden

Häufig kommt es vor, dass durch Hyperlinks mehrere Fenster der gleichen Art geöffnet werden sollen. Um dies zu realisieren, gibt es verschiedene Möglichkeiten. Im schlechtesten Fall würde für jeden Aufruf eine eigene Funktion definiert werden. Dies ist jedoch nicht nur nicht elegant, sondern auch zeit- und ladeintensiv. Eine bessere Lösung bietet hier die Möglichkeit, JavaScript-Funktionen mit Parametern zu definieren. Wie sich dies realisieren lässt, beschreibt das folgende Beispiel. In der Datei werden drei Grafiken angezeigt. Durch Anklicken der Grafiken wird jeweils ein anderer URL in einem neuen Fenster geöffnet. Das Besondere hieran ist, dass nicht drei Funktionen definiert werden müssen. Das Beispiel kommt, durch die Parameter-Variante, mit nur einer JavaScript-Funktion aus.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Musik(URL)
{
Fenster = window.open(URL,'Bands','width=240,height=250,left=0,top=0,
scrollbars=no')
Fenster.focus()
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Musik('http://www.offspring.com')"><IMG src="grafik1.gif"
width="50" height="50"></A><BR>
<A href="javascript:Musik('http://www.bosstones.com')"><IMG src="grafik2.gif"
```

```
width="50" height="50"></A><BR>
<A href="javascript:Musik('http://www.greenday.com')">
<IMG src="grafik3.gif" width="50" height="50"></BR>
</BODY
</HTML>
```

Listing 5.31: Das Öffnen mehrerer Fenster durch Parameterübergabe

Das Anklicken der Grafiken bewirkt das Auslösen der `Musik()`-Funktion. Als Parameter wird jedem Funktionsaufruf ein anderer URL zugewiesen. Dies ist möglich, da die Funktion `Musik()` mit einem Parameter definiert wurde. Innerhalb der Funktion wird der Variablen `Fenster` die **`open()`**-Methode zugewiesen. In dieser Methode werden zusätzlich die Eigenschaften des neuen Fensters festgelegt. Neben dem Fensternamen `Bands` werden hier eine Breite von `240` und eine Höhe von `250` definiert. Die weiteren Eigenschaften beziehen sich auf das weitere Erscheinungsbild der Seite. Also die Startposition von oben und von links und dass bei dem Fenster keine Scrollbalken gestattet sind. Das wichtigste Element ist der Parameter `URL`. Dieser wird als Erster innerhalb der **`open()`**-Methode notiert und legt fest, dass es sich hierbei um den anzuzeigenden URL handelt. Welcher URL letztendlich aufgerufen werden soll, wird über den Aufruf der Funktion durch den Hyperlink festgelegt. Achten Sie darauf, dass der verwendete URL dort in Anführungszeichen gesetzt notiert werden muss.

5.31 Neue Seite beim Verlassen der Seite

eine nervige Angelegenheit

Das hier vorgestellte Beispiel bedient sich eines zu Recht umstrittenen Stilmittels. Diese Syntax ermöglicht es, dass nach dem Verlassen der aktuellen Seite ein neues Fenster geöffnet wird. Eingesetzt wird dies vor allem im Zusammenhang mit Werbebotschaften. Selbstverständlich ist es so, dass der Anwender von sich selbstständig öffnenden Fenstern nicht begeistert ist. Dennoch habe ich mich dazu entschlossen, auch ein solches Beispiel hier vorzustellen. Schließlich ist es ja so, dass Sie als Entwickler häufig Kundenwünschen unterliegen und diese in vielen Fällen solche Werbemaßnamen wünschen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Werbung(){
window.open("banner.htm", "newWin", "toolbar=no,location=no,
directories=no,status=no, menubar=no,scrollbars=no,resiz-
able=no,left=0,top=0,width=420,height=130")
```

```
}
//-->
</SCRIPT>
</HEAD>
<BODY onunload="Werbung()">
</BODY>
</HTML>
```

Listing 5.32: Die Datei `banner.htm` wird in einem neuen Fenster angezeigt.

Der Event-Handler **`onunload`** wird dann aktiviert, wenn die aktuelle Seite verlassen wird, und löst die Funktion `Werbung()` aus. Hierin wird über die **`open()`**-Methode des **`window`**-Objekts ein neues Fenster geöffnet. Als URI wird `banner.htm` angegeben. Zudem werden der Name `newWin` sowie Eigenschaften wie Breite und Höhe, Startposition von links und oben definiert. Weitere Gestaltungsmittel wie das Ausblenden der Menüleiste u.Ä. werden hier ebenfalls festgelegt.

5.32 Favoriten im IE hinzufügen

schnellerer Seitenzugriff

Der Internet Explorer ermöglicht einen schnellen Zugriff auf die Lieblingsseiten des Anwenders. Microsoft bedient sich hierbei des Prinzips der Favoriten. Der Anwender kann anhand dieser Technologie seine Lieblingsseiten per Hand in eine Liste aufnehmen und diese anschließend wieder aufrufen. Zwar bietet das hier gezeigte Script keineswegs die Garantie, dass Ihre Seiten auch den Weg in die Favoritenliste vieler Anwender findet, es ist allerdings ein guter Weg, dem Nutzer diese Entscheidung zu vereinfachen. Die folgende Syntax beschreibt, wie ein Textlink dazu verwendet werden kann, einen bestimmten URL zu der Favoritenliste des Anwenders hinzuzufügen.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:window.external.
AddFavorite('http://www.myhost.com',
'Prima Seite')">Zu den Favoriten</A>
</BODY>
</HTML>
```

Listing 5.33: Diese Syntax funktioniert nur im Internet Explorer.

Um einen Eintrag zur Favoritenliste hinzuzufügen, muss die Anweisung **`window.external.AddFavorite`** notiert werden. Als Parameter werden zunächst der URL der Seite sowie ein beschreibender Text benötigt. Es

sollte darauf geachtet werden, dass der hierbei vergebene Text nicht das Maß des Erträglichen überschreitet. Der Internet Explorer unterstützt als einziger Browser die hier gezeigte Syntax. Für Ihre Besucher sollte demzufolge ein entsprechender Hinweistext vorhanden sein.

5.33 Popup-Generator

Der Anwender kann sein eigenes Fenster generieren.

Lassen Sie doch den Anwender selbst entscheiden, wie ein neu zu öffnendes Fenster aussehen soll. Dies lässt sich ganz einfach mit dem hier gezeigten Beispiel realisieren. Dem Anwender wird ein Popup-Generator angeboten. Mit diesem können das Aussehen sowie der Ziel-URL bestimmt werden. Bezüglich des Erscheinungsbildes des neuen Fensters kann die Darstellung fast aller Elemente modifiziert werden. So lassen sich neben der Fenstergröße auch die Menüleiste und die Scrollbalken ein- bzw. ausschalten. Dieses Beispiel soll zudem verdeutlichen, wie sich Automatismen in JavaScript realisieren lassen. So könnte eine ähnliche Syntax auch für die Darstellung von Grafiken verwendet werden. Die nachstehende Syntax beinhaltet einzeilige Eingabefelder, Checkboxen sowie einen Formular-Button. In den Eingabefeldern werden der Fenstername, der Ziel-URL sowie die Breite und Höhe des neuen Fensters angegeben. Die Checkboxen werden benutzt, um zu bestimmen, ob das neue Fenster mit Bildlaufleisten usw. versehen werden soll. Wobei hier das Aktivieren der Checkboxen zur Anzeige des gewünschten Fensterelements führt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function PopupGenerator()
{
var URL = document.Formular.URL.value;
var Feld = document.Formular;
var Fenstername= Feld.Fenstername.value;
var Eigenschaften =
        'width='        + Feld.Breite.value +
        ',height='      + Feld.Hoehe.value +
        ',status='      + (Feld.Statuszeile.checked - 0) +
        ',toolbar='     + (Feld.Werkzeugleiste.checked - 0) +
        ',location='    + (Feld.Adresszeile.checked - 0) +
        ',menubar='     + (Feld.Menueleiste.checked - 0) +
        ',scrollbars='  + (Feld.Bildlaufleisten.checked - 0) +
        ',resizable='   + (Feld.veraenderbar.checked - 0);
    window.open (URL, Fenstername, Eigenschaften);
}
```

```
//-->
</SCRIPT>
</HEAD>
<BODY>
<Form name="Formular">
URL: <INPUT type="text" name="URL" value="neues.htm">
Name: <INPUT type="text" name="Fenstername" value="Fenstername">
Breite: <INPUT type="text" name="Breite" value="300">
Höhe: <INPUT type="text" name="Hoehe" value="250">
Bildlaufleisten: <INPUT type="checkbox" name="Bildlaufleisten">
Statuszeile: <INPUT type="checkbox" name="Statuszeile">
Adresszeile: <INPUT type="checkbox" name="Adresszeile">
Menueleiste: <INPUT type="checkbox" name="Menueleiste">
Werkzeugleiste: <INPUT type="checkbox" name="Werkzeugleiste">
Größenvariabel: <INPUT type="checkbox" name="veraenderbar">
<INPUT type="button"
value="generieren"onclick="PopupGenerator()">
</FORM>
</BODY>
</HTML>
```

Listing 5.34: Das neue Fenster kann recht frei gestaltet werden.

Durch Anklicken des Formular-Buttons wird die Funktion `PopupGenerator()` ausgelöst. Diese verwendet die **`open()`**-Methode, um das neue Fenster anzulegen. Um die Eigenschaften des neuen Fensters zu definieren, werden Variablen verwendet. Deren Werte ergeben sich aus den Werten der Eingabefelder bzw. Checkboxen. Ohne Variablen würde die `PopupGenerator()`-Funktion folgende Syntax besitzen:

```
<SCRIPT type="text/JavaScript">
<!--
function PopupGenerator()
{
window.open ("bneu.htm", "Fenstername", width=300,height=200, status=0,
toolbar=0, location=0, menubar=0, scrollbars=0, resizable=0);
}
//-->
</SCRIPT>
```

Alle Parameter der **`open()`**-Methode, die hier fest kodiert sind, werden in dem Beispiel durch Variablen definiert. Als Beispiel sei hier die Breite des Fensters genannt. Die Breite wird über die **`width`**-Eigenschaft bestimmt. Um den entsprechenden Wert aus dem Eingabefeld auszulesen, wird auf den Wert des Feldes `Breite` über die Anweisung `Feld.Breite`**`.value`** zugegriffen. Bei Checkboxen verhält sich dies etwas anders. Soll das neue Fenster mit einer Menüleiste versehen werden, muss die entsprechende

Checkbox „angekreuzt“ werden. Um den Zustand der Checkbox zu überprüfen, wird über `Feld.Menueleiste.checked` auf die Checkbox zugegriffen. Die Anweisung `–0`, die bei jeder Checkbox des Beispiels vordefiniert ist, besagt, dass die Checkbox nicht aktiviert ist. Wird die Checkbox „angekreuzt“ verändert sich dieser Wert in `–1` und die Menüleiste wird im neuen Fenster mit angezeigt.

5.34 Fragen und Übungen

1. Welches sind gültige Eigenschaften des `window`-Objekts?

   ```
   closed
   stopp()
   statusbar
   bgcolor
   ```

2. Schreiben Sie eine Funktion, durch die beim Einlesen einer Seite ein zweites Fenster geöffnet wird. Im Hauptfenster soll sich ein Verweis befinden. Durch dessen Anklicken soll in einem Meldungsfenster ausgegeben werden, ob das neue Fenster noch offen ist oder nicht.

6 Dokumente

lernen

Ziele dieses Kapitels

Nachdem Sie nun den Umgang mit dem Browserfenster kennen gelernt haben, widmen wir uns in diesem Kapitel dem **document**-Objekt. Mit dem gewonnenen Wissen können Sie u.a. Inhalte dynamisch in das Dokument schreiben. Als weitere Eckpfeiler werden wir einige Elemente des Document Object Model behandeln. Hierbei wird der Fokus vor allem auf den Zugriff auf Elemente sowie die Knoten eines Dokuments gerichtet.

6.1 Zugriff auf das Dokument

Document Object Model

Durch das **document**-Objekt haben Sie Zugriff auf den Inhalt des Browserfensters. In der JavaScript-Objekthierarchie liegt das **document**- direkt unter dem **window**-Objekt. Das World Wide Web Consortium (W3C) nutzt das **document**-Objekt als Ausgangspunkt für den Elementbaum des Document Object Model (DOM). Was ist ein solcher Elementbaum? HTML-Dateien bestehen aus Tags, Attributen und Wertzuweisungen. Die HTML-Variante des DOM besagt nun, dass jedes Element einer HTML-Datei ein eigenes Objekt ist. Jedes HTML-Objekt stellt innerhalb des Elementbaums einen Knoten dar. Knoten können ein Kindelement oder ein Elternelement sein. So besagt die Syntax **<P><DIV></DIV></P>**, dass **<DIV>** ein Kindelement von **<P>** und **<P>** das Elternelement von **<DIV>** ist. Mehr zu diesem Thema erfahren Sie im Laufe dieses Abschnitts. Um auf die Eigenschaften und Methoden des **document**-Objekts zuzugreifen, können die beiden folgenden Syntaxformen verwendet werden:

```
window.document.Eigenschaft
window.document.Methode()
```

Hinter dem **window**- wird das **document**-Objekt notiert. Dies ergibt sich aus der JavaScript-Objekthierarchie. Eigenschaften und Methoden werden im Zusammenhang mit dem **document**-Objekt folgendermaßen verwendet:

```
window.document.charset="iso-8859-5";
window.document.close();
```

Bezieht sich die Angabe einer Eigenschaft oder Methode auf das aktuelle Fenster, kann auf das Notieren des **window**-Objekts verzichtet werde. Den gleichen Effekt wie zuvor würden Sie also auch mittels der nachstehenden Syntax erreichen.

```
document.charset="iso-8859-5";
document.close();
```

Anders verhält es sich bei dem Zugriff auf Dokumentinhalte von anderen Fenstern. In einem solchen Fall muss der reale Name des Fensters angegeben werden. Wurde beispielsweise ein Fenster über die **open()**-Methode geöffnet und diesem der Name **neuesFenster** zugewiesen, ergäbe sich hieraus die folgende Syntax:

```
neuesFenster.document.charset="iso-8859-5";
neuesFenster.document.close();
```

Einen ersten Einblick in die Handhabung des **document**-Objekts, dessen Eigenschaften und Methoden erhalten Sie mittels des folgenden Beispiels. Hierin wird über die **prompt()**-Methode ein Dialogfenster geöffnet. Der Anwender wird aufgefordert, seinen Namen einzugeben. Nach der Bestätigung der Eingabe wird der Anwender persönlich begrüßt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
Name = prompt("Geben Sie bitte Ihren Namen an:","Ihr Name");
document.write("<i>Hallo " + Name + "<\/i>");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.1: Die Generierung eines Dialogfensters

Der Wert des Dialogfeldes wird in der Variablen `Name` gespeichert. Um den eingegebenen Namen in das Dokument zu schreiben, wird die **write()**-Methode des **document**-Objekts verwendet.

Abbildung 6.1: Ein Dialogfeld mit vordefiniertem Inhalt

6.2 In das Dokument schreiben

Inhalte dynamisch anzeigen

Die **write()**-Methode des **document**-Objekts ermöglicht es, einen beliebigen Text in eine Datei zu schreiben. Erlaubt ist der Einsatz der **write()**-Methode seit der JavaScript-Version 1.0 und wird sowohl vom Internet Explorer wie auch vom Netscape Navigator unterstützt. Der **write()**-Methode muss eine Zeichenkette, bzw. ein geeigneter JavaScript-Ausdruck zugewiesen werden. Sinnvoll ist diese Syntax vor allem hinsichtlich des dynamischen Anzeigens von Textinhalten. Ein Beispiel:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
document.write("Das ist die Datei "+document.URL);
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.2: Der URL der Datei wird in das Dokument geschrieben.

Hier wird innerhalb des Dateikörpers eine **document.write()**-Anweisung notiert. Deren Ausführung findet augenblicklich mit dem Aufrufen der Datei statt. Innerhalb des Browsers wird dem Anwender der URL der gerade aufgerufenen HTML-Datei mitgeteilt. Zu beachten ist die gesonderte Notationsform im Hinblick auf JavaScript-Ausdrücke, die der **write()**-Methode als Parameter zugewiesen werden sollen. Während normale Zeichenketten lediglich in Anführungszeichen gesetzt werden, kommt der Notation eines JavaScript-Ausdrucks eine gesonderte Syntax zu, nämlich eine ohne Anführungszeichen.

Anweisungen kombinieren

Für die Entwicklung tatsächlich praxistauglicher Anwendungen erscheint das Prinzip der Verknüpfung von String und Ausdruck innerhalb der **write()**-Methode das probateste Mittel. Wurde dieses Prinzip bereits im ersten Beispiel anhand einer solchen Verknüpfung gezeigt, soll die folgende Syntax eine doppelte Kombination aus jeweils zwei Strings und zwei JavaScript-Ausdrücken veranschaulichen.

```
<HTML>
<HEAD>
<TITLE>Datei-Titel</TITLE>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
document.write("Das ist die Datei "+document.URL+ " mit dem Titel "
+document.title);
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.3: Der URL und der Titel der Datei werden in das Dokument geschrieben.

Das Beispiel zeigt, wie innerhalb der aktuellen Datei ein hinweisender Text dynamisch generiert wird. Dieser gibt neben statischen Strings auch den URL sowie den Datei-Titel über die **write()**-Methode aus.

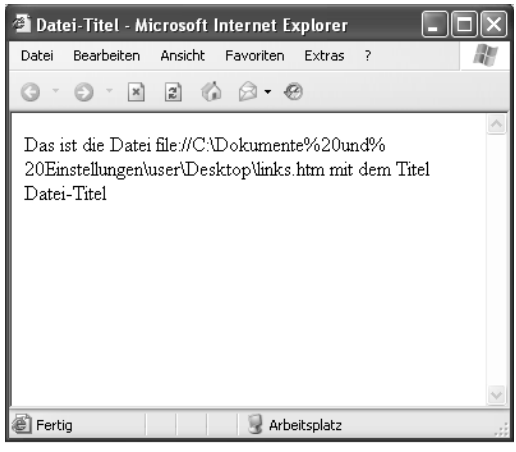


Abbildung 6.2: Der Datei-Titel wird dynamisch in das Dokument geschrieben.

6.3 Zeilenweise schreiben

***Unterschiede und Gemeinsamkeiten zwischen* write() *und* writeln()**

Prinzipiell handelt es sich bei der **writeln()**- um eine sich in ihrem Verhalten zur **write()**-Methode sehr ähnlich verhaltende JavaScript-Anweisung. Und dennoch sind einige Besonderheiten zu beachten. So eignet sich die **write()**-Methode zwar vortrefflich für die Generierung von Fließtexten, im Hinblick auf Anwendungen, die den Einsatz zahlreicher Zeilenumbrüche erfordern, ist deren Verwendung allerdings nur bedingt geeignet. Zwar lassen sich in JavaScript Zeilenumbrüche problemlos über die Zeichenfolge \n realisieren, aber gerade bei umfangreichen Scripts kann diese Vorgehensweise die Übersichtlichkeit des Quelltextes schmälern. In einem solchen Fall ist die **writeln()**-Methode die bessere

Variante. Durch deren Einsatz werden die Informationen zeilenweise in das Dokument geschrieben. Ein Beispiel:

```
<HTML>
<HEAD>
<TITLE>Writeln</TITLE>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
neu = window.open("","");
neu.document.open("text/plain");
neu.document.writeln("Zeichensatz: "+document.charset);
neu.document.writeln("Datei-Titel: " + document.title);
neu.document.writeln("Datei-Titel: " + document.URL);
neu.document.close();
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.4: Ein neues Fenster mit vordefiniertem Inhalt

In dem Beispiel wird zunächst ein neues Fenster geöffnet. Hierin werden zeilenweise der verwendete Zeichensatz, der Dateititel sowie der URL der Datei in das Dokument geschrieben.

6.4 Fensterinhalte freigeben

Die **open()**-Methode kennen Sie bereits vom **window**-Objekt. Doch während dort die **open()**-Methode ein neues Fenster öffnet, öffnet die **open()**-Methode des **document**-Objekts ein neues Dokument. Der Unterschied zwischen den beiden gleichnamigen Methoden besteht also darin, dass im Zusammenhang mit dem **document**-Objekt der Fensterinhalt zum Neubeschreiben freigegeben wird. Sie können die **open()**-Methode mit keinem, mit einem oder mit zwei Parametern verwenden. Die folgenden beiden Parameter sind einsetzbar:

mögliche Parameter

- MIME-Typ – Gibt die Art der Datei an. Soll eine HTML-Datei erzeugt werden, muss hier **text/html** notiert werden.
- **replace** – Die neu erzeugte Datei nimmt den Platz der alten Datei in der History des Browsers ein.

Das folgende Beispiel zeigt, wie ein neues Dokument geöffnet und mit definierten Inhalten gefüllt werden kann. Besonderes Augenmerk ist hierbei auf die **back()**-Methode des **history**-Objekts zu legen. Dieses dient dazu, auf die Seite zu springen, die in der History vor der aktuellen steht. Diese Methode funktioniert in diesem Beispiel allerdings nicht wie ge-

dacht. Durch den Einsatz des **replace**-Parameters wird ja die alte Seite durch das neue Dokument überschrieben. Die alte Seite ist somit in der History des Browsers nicht mehr vorhanden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
function neuerInhalt()
{
document.open("text/html","replace");
document.write("<A href=\"javascript:history.back()\">zurück</A>");
document.close();
}
document.open("text/html","replace");
document.write("<A href=\"javascript:neuerInhalt()\">alter Inhalt</A>");
document.close();
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.5: Der Zurück-Effekt funktioniert hier nicht.

Der Browser führt zunächst den unteren Teil des JavaScript-Codes aus. Der obere Teil befindet sich innerhalb einer Funktion. Das Anklicken des Hyperlinks löst die Funktion **neuerInhalt()** aus. In dieser Funktion wird über die **open()**-Methode ein neues Dokument geöffnet. Da dessen Inhalt aus HTML-Code besteht, wird der MIME-Typ **text/html** angegeben. Damit die alte Seite überschrieben wird, wird der **replace**-Parameter notiert. Durch das Auslösen der Funktion wird der Inhalt des aktuellen Dokuments mit dem neuen Inhalt überschrieben. An Stelle des Textes `alter Inhalt` wird dem Anwender der Hyperlink `zurück` angezeigt. Wie gesagt, führt dieser allerdings nicht zu der gewünschten Seite, nämlich dem alten Dokumentinhalt.

6.5 Farbe des aktiven Hyperlinks

möglicher Browserkonflikt

Über die Eigenschaft **alinkColor** wird der Farbwert gespeichert, der dem Attribut **alink** im einleitenden **<BODY>**-Tag zugewiesen wurde. Das **alink**-Attribut dient der dateiweiten Definition der Farbe des aktuell angeklickten Hyperlinks. Beachten Sie, dass diese Farbe nicht in jedem Fall vom eingesetzten Browser akzeptiert werden muss. Der Anwender hat die Möglichkeit, seine eigene Farbdefinition für den gerade aktivierten

Hyperlink einzustellen. Hierzu müssen lediglich einige Modifikationen innerhalb der Browsereinstellungen vorgenommen werden. Nimmt der Anwender diese Einstellungen vor, innerhalb der aufgerufenen Seite befindet sich jedoch ein **alink**-Attribut mit einem anderen Farbwert, kommt es zum Konflikt. Dieser Konflikt wird vom Browser stets so gehandhabt, dass die Nutzereinstellungen bevorzugt behandelt werden. Um über das **alink**-Attribut die Darstellung von gerade angeklickten Verweisen zu definieren, können sowohl Farbwörter wie auch hexadezimale Werte notiert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function wechselFarbe()
{
 if(document.Formular.test[0].checked == true)
 document.alinkColor = "red";
 else if(document.Formular.test[1].checked == true)
 document.alinkColor = "blue";
 else document.alinkColor = "000000";
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="radio" value="rot" name="test">
<INPUT type="radio" value="blau" name="test">
</FORM>
<A href="javascript:wechselFarbe()">Linkfarbe &auml;ndern</A>
</BODY>
</HTML>
```

Listing 6.6: Die Linkfarbe kann frei gewählt werden.

Das Beispiel enthält ein Formular, in welchem sich zwei Radio-Buttons befinden. Unterhalb hiervon wurde ein Hyperlink notiert, durch dessen Anklicken die Funktion `wechselFarbe()` aufgerufen wird. In dieser wird in Form einer **if**-Abrage überprüft, welcher der beiden Radio-Buttons selektiert ist. Handelt es sich hierbei um den ersten, wird der gerade angeklickte Hyperlink in roter Farbe angezeigt. Bei der Selektierung des zweiten Radio-Buttons wird der aktuell angeklickte Verweis in blauer Farbe dargestellt.

6.6 Hintergrundfarbe

Kontrast zwischen Hintergrund- und Textfarbe erhalten!

Mit der Eigenschaft **bgcolor** kann die Hintergrundfarbe, welche innerhalb des **<BODY>**-Tags spezifiziert wurde, ausgelesen und verändert werden. Ebenso wie auf diese Farbwertbestimmung kann auch auf die vom Anwender innerhalb seiner Browsereinstellungen vorgenommene Hintergrundeigenschaft zugegriffen werden. Das alleinige Auslesen der Hintergrundfarbe einer HTML-Datei macht freilich nur bedingt Sinn. Als Ausnahme kann hierbei jedoch die Verwendung der **bgcolor**-Eigenschaft innerhalb einer **if**-Abfrage dienen. Anspruchsvoller in der Ausführung, wenn auch simpel in der Programmierung, beschreibt das folgende Beispiel, wie die Hintergrundfarbe einer HTML-Datei per HTML-Button variabel gestaltet werden kann.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
   function Farbwert(Wert)
   {
      document.bgColor=Wert;
   }
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT type="button" value="Weiss" onclick="Farbwert('ffffff');">
<INPUT type="button" value="Schwarz" onclick="Farbwert('000000');">
<INPUT type="button" value="Grau" onclick="Farbwert('AAAAAA');">
<INPUT type="button" value="Blau" onclick="Farbwert('0000ff');">
</FORM>
</BODY>
</HTML>
```

Listing 6.7: Die Hintergrundfarbe ist modifizierbar.

kein Doppelkreuz bei hexadezimalen Farbwerten

Innerhalb des aufgeführten Beispiels sind vier Formular-Buttons definiert, durch deren Anklicken sich die Hintergrundfarbe der aktuellen Datei verändern lässt. Besonders geeignet ist eine solche Anwendung vor allem im Hinblick auf eine für unterschiedliche Nutzerbedürfnisse optimierte Seite. So könnte beispielsweise auf jeder Seite eines Internetauftritt dem Anwender die Möglichkeit zur Einstellung der für ihn bestmöglichen Hintergrundfarbe geboten werden. Und eben für einen solchen Einsatz scheint das gezeigte Script geradezu prädestiniert zu sein. In dem Beispiel wird durch jedes Anklicken eines Buttons dem Doku-

menthintergrund ein definierter Farbwert zugewiesen. Realisiert wird dies durch eine Kombination aus der Funktion `Farbwert()` und dem Event-Handler **`onclick`**. Hinsichtlich der Definition von Hintergrundfarben ist lediglich auf deren doch von der üblichen Schreibweise in HTML abweichende Notation zu achten. Zwar stehen auch in JavaScript die Varianten Farbnamen und hexadezimale Farbwerte zur Verfügung, im Hinblick auf zuletzt Genannte ist jedoch eine gesonderte Notationsform zu berücksichtigen. Anders, als dies in HTML üblich ist, wird hexadezimalen Farbwerten kein Doppelkreuz vorangestellt. Aus der üblichen Schreibweise für einen weißen Hintergrundrund *#ffffff* wird demnach bei der Definition einer Hintergrundfarbe in JavaScript *ffffff*. Darüber hinaus ist darauf zu achten, dass in der Regel Farbwerte, und dies gilt sowohl für Farbnamen wie auch für hexadezimale Angaben, in Anführungszeichen gesetzt werden müssen. Als Ausnahme von dieser Regel gelten all jene Anwendungen, die sich der dynamischen Generierung von Farbwerten bedienen.

6.7 Zeichensatz

multinationale Internetprojekte

WWW-Seiten werden längst nicht nur von Anwendern aus ein und demselben Land bzw. Sprachraum betrachtet. Unterschiedliche Herkunftsländer bedeuten für den Seitenentwickler jedoch häufig einen zusätzlichen Aufwand. Dies ist vor allem dann von Bedeutung, wenn die jeweiligen Anwender aus einem anderen Sprachkreis kommen und somit in ihrem Browser einen anderen Zeichensatz eingestellt haben. Zeichensätze sind ein recht komplexes Thema, welches weit über das hier zu vermittelnde Grundwissen hinausgeht. Deswegen an dieser Stelle nur so viel: Die Verwendung unterschiedlicher Zeichensätze führt dazu, dass verschiedene Zeichen bei den jeweiligen Anwendern anders oder falsch interpretiert werden. So können Sie zwar beispielsweise ziemlich sicher sein, dass im deutschen Sprachraum die Verwendung von Umlauten zu keinerlei Anzeigeproblemen führt, im kyrillischen Sprachkreis verhält sich dies dann jedoch anders. Um diesem Umstand entgegenzuwirken, können Sie über die **`charset`**-Eigenschaft den verwendeten Zeichensatz auslesen und ändern. Diese Möglichkeit nutzt das nachstehende Beispiel.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
if (document.referrer == 'http://www.yahoo.ru')
document.charset="iso-8859-5";
```

```
else  document.charset="iso-8859-1";
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.8: Der Zeichensatz wird der Herkunft der Suchmaschine angepasst.

Beim Einlesen der Datei wird anhand einer **if**-Abfrage überprüft, ob der Referrer der URI *http://www.yahoo.ru* ist. Wird diese Bedingung erfüllt, findet der für den kyrillischen Sprachkreis entwickelte Zeichensatz **iso-8859-5** Verwendung. Anderenfalls wird der für den westeuropäischen Sprachraum geschaffene **iso-8859-1** Zeichensatz angewandt. Selbstverständlich ließe sich diese Syntax noch weiter ausbauen und somit für alle gängigen Sprachräume erweitern.

6.8 Standard-Zeichensatz

Bezug zwischen Zeichensatz und Herkunft des Anwenders

Die **defaultcharset**-Eigenschaft speichert den beim Anwender als Standard definierten Zeichensatz. Für den Fall, dass die aufgerufene Datei keine dem Zeichensatz bezügliche Meta-Angabe besitzt, wird der Standardzeichensatz verwendet. Da man mit ziemlicher Sicherheit davon ausgehen kann, dass der beim Anwender als Standard eingestellte Zeichensatz einen Bezug zu dessen Herkunft gestattet, kann die **defaultCharset**-Eigenschaft vorzüglich für die Weiterleitung des Anwenders auf eine für ihn optimierte Seite genutzt werden. Das gilt freilich nur in dem Umfang, da es sich um unterschiedliche Sprachräume mit verschiedenen Zeichensätzen handelt. So lässt sich also beispielsweise sehr wohl unterscheiden, ob der Anwender aus dem russischen oder deutschen Sprachraum stammt. Für eine Unterscheidung zwischen einem Anwender aus dem italienischen und französischen Sprachraum ist die **defaultCharset**-Eigenschaft indes nicht geeignet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
if(document.defaultCharset = "iso-8859-1")
 location.href = "german.htm";
else if(document.defaultCharset = "iso-8859-5")
 location.href = "kyr.htm";
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 6.9: Je nach eingestelltem Zeichensatz wird eine andere Seite angezeigt.

Beim Aufrufen der Seite wird der Script-Bereich ausgeführt. Hierin wird anhand einer **if**-Abfrage überprüft, welcher Standardzeichensatz im Browser des Anwenders eingestellt ist. Handelt es sich um **iso-8859-1,** wird der Nutzer auf die Seite `german.htm` weitergeleitet. Es handelt sich zwar hierbei nicht zwingend um einen Deutsch sprechenden Zeitgenossen, die Chance hierfür ist aber dennoch gegeben. Sollte hingegen als Standardzeichensatz **iso-8859-5** eingestellt sein, folgt die automatische Weiterleitung auf die Seite `kyr.htm`. Für die Weiterleitung wird in beiden Fällen die **href**-Eigenschaft des **location**-Objekts verwendet.

6.9 Textfarbe

Browsereinstellungen vor JavaScript

Die **fgcolor**-Eigenschaft speichert die Farbe des Textes, wie sie innerhalb des einleitenden **<BODY>**-Tags anhand des **text**-Attributs oder durch den Anwender in den Browsereinstellungen definiert wurde. Der Vorteil der Zuweisung einer dateiweiten Textfarbe liegt auf der Hand. Muss doch dank dieser Anweisung die Farbdefinition nicht permanent im Quelltext wiederholt werden. Für den Fall, dass ein **text**-Attribut notiert wurde, die Textfarbe in den Browsereinstellungen jedoch eine andere ist, tritt ein Konflikt auf. In einem solchen Fall werden die Einstellungen des Anwenders denen des Entwicklers vorgezogen. Die Farbgebung des **text**-Attributs wird demnach nicht berücksichtigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Farbwert()
{
 if(document.Formular.Farbe[0].checked == true)
 document.fgColor = "red";
 else if(document.Formular.Farbe[1].checked == true)
 document.fgColor = "blue";
 else if(document.Formular.Farbe[2].checked == true)
 document.fgColor = "green";
 else document.alinkColor = "000000";
}
</SCRIPT>
</HEAD>
<BODY>
<P>Was m&ouml;gen Sie am liebsten?</P>
<FORM name="Formular">
Rot: <INPUT type="radio" value="rot" name="Farbe">
Blau: <INPUT type="radio" value="blau" name="Farbe">
Grün: <INPUT type="radio" value="gruen" name="Farbe">
</FORM>
```

```
<A href="javascript:Farbwert()">Textfarbe &auml;ndern</A>
</BODY>
</HTML>
```

Listing 6.10: Die Textfarbe kann dynamisch verändert werden.

Beim Laden der Seite werden ein Text und drei Radio-Buttons angezeigt. Der Anwender kann die Farbe des angezeigten Textes nachträglich ändern. Hierzu wird die Funktion `Farbwert()` genutzt, welche durch Anklicken des Verweises ausgelöst wird. Diese Funktion überprüft zunächst in Form einer **`if`**-Abfrage, welcher der drei Radio-Buttons vom Anwender selektiert wurde. Anschließend greift sie auf die Farbe des dateiweiten Textes zu. Hierfür wird die Anweisung **`document.fgColor`** verwendet. Für die jeweilige Farbgebung wird ein entsprechender Farbwert notiert. Wie das Beispiel zeigt, sind hierbei sowohl Farbnamen wie auch hexadezimale Werte zulässig.

6.10 Letzte Änderung des Dokuments

Ausgabe in englischer Sprache und im GMT-Format

Zu welchem Zeitpunkt eine Datei modifiziert wurde, lässt sich anhand der Eigenschaft **`lastModified`** anzeigen bzw. dynamisch in das Dokument schreiben. Eingesetzt wird **`lastModified`** vor allem im Zusammenhang mit Internetprojekten, deren Reiz für den Anwender eben gerade auf häufig aktualisierten datumsabhängigen Informationen beruht. Das folgende Beispiel zeigt, wie sich durch das Zusammenspiel der Eigenschaft **`lastModified`** mit der Methode **`write()`** der Zeitpunkt der letzten Änderung des aktuellen Dokuments anzeigen lässt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
document.write("Letzte &Auml;nderung: " + document.lastModified);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.11: Die Verwendung der `lastModified`-Eigenschaft

Beachten Sie, dass die hierdurch eingefügte Datumsanzeige im Greenwich-Zeitschema angezeigt wird. So würde die Anzeige 03/25/2002 beispielsweise den 25. März 2002 bedeuten. Zusätzlich zum Datum wird die exakte Uhrzeit der letzen Modifizierung sekundengenau mit angezeigt.

Abbildung 6.3: Der Zeitpunkt der letzten Modifikation der Datei

6.11 Linkfarbe

gleiche Farbwerte für alle Hyperlinks

Die `linkColor`-Eigenschaft speichert den Farbwert für Verweise, wie dieser innerhalb des einleitenden `<BODY>`-Tags durch das `link`-Attribut bzw. durch die Browsereinstellungen des Anwenders festgelegt wurde. Als mögliche Angaben können hierbei sowohl hexadezimale Werte wie auch Farbwörter verwendet werden. Von Vorteil ist die Verwendung des `link`-Attributs immer dann, wenn innerhalb der aktuellen Datei mehrere Hyperlinks vorkommen, die in der gleichen Farbe dargestellt werden sollen. Hier muss dann die Farbzuweisung nur einmal und nicht, wie sonst üblich, bei jeder Hyperlinkdeklaration vorgenommen werden. Sie sollten dennoch bedenken, dass die Definition der Hyperlinkfarbe nicht in jedem Fall Gültigkeit besitzt. Sollte der Anwender innerhalb der Browsereinstellungen eine bestimmte Linkfarbe als Standardwert eingestellt haben, wird diese in jedem Fall der auf der Seite befindlichen Farbdefinition vorgezogen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function LinkFarbe()
{
 if(document.Formular.Farbe[0].checked == true)
 document.linkColor = "red";
 else if(document.Formular.Farbe[1].checked == true)
 document.linkColor = "blue";
 else if(document.Formular.Farbe[2].checked == true)
 document.linkColor = "green";
 else document.linkColor = "000000";
}
//-->
</SCRIPT>
```

```
</HEAD>
<BODY>
<P>Was m&ouml;gen Sie am liebsten?</P>
<FORM name="Formular">
Rot: <INPUT type="radio" value="rot" name="Farbe">
Blau: <INPUT type="radio" value="blau" name="Farbe">
Grün: <INPUT type="radio" value="gruen" name="Farbe">
</FORM>
<A href="javascript:LinkFarbe()">Linkfarbe &auml;ndern</A>
<BR>
<A href="#">Testlink</A>
</BODY>
</HTML>
```

Listing 6.12: Frei wählbare Linkfarbe

Dem Anwender werden anhand der aufgeführten Syntax drei Radiobuttons sowie ein Hyperlink angezeigt. Er kann hierdurch die Farbe des Hyperlinks auf dieser Seite selbst auswählen. Möglich sind hierbei die drei Farben Rot, Blau und Grün. Durch Anklicken des Verweises wird die Funktion `LinkFarbe()` ausgelöst. Diese überprüft zunächst in Form einer **`if`**-Abfrage, welcher der drei Radiobuttons selektiert wurde. Anschließend wird über **`document.linkColor`** auf die Linkfarbe zugegriffen. Je nachdem, welcher Radiobutton ausgewählt ist, wird ein entsprechender Farbwert angewendet.

6.12 Farbe von besuchten Links

hexadezimale Werte und Farbwörter

Gerade im Hinblick auf Seiten, in denen zahlreiche Hyperlinks definiert sind, fällt der **`vlinkColor`**-Anweisung eine enorme Bedeutung zu. Im Ergebnis handelt es sich hierbei um die gleiche Anweisung wie bei dem Attribut **`vlink`**, welches innerhalb des **`<BODY>`**-Tags eingesetzt wird. Denn auch über **`vlinkColor`** wird die Farbe derjenigen Hyperlinks definiert, die auf Seiten zeigen, welche vom Anwender bereits besucht wurden. Die Eigenschaft **`vlinkColor`** erwartet eine Farbangabe. Hierbei können sowohl hexadezimale Werte wie auch Farbnamen verwendet werden. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Linkfarbe()
{
 if(document.Formular.Farbe[0].checked == true)
 document.vlinkColor = "red";
```

```
 else if(document.Formular.Farbe[1].checked == true)
 document.vlinkColor = "blue";
 else if(document.Formular.Farbe[2].checked == true)
 document.vlinkColor = "green";
 else document.alinkColor = "000000";
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P>Was m&ouml;gen Sie am liebsten?</P>
<FORM name="Formular">
Rot: <INPUT type="radio" value="rot" name="Farbe">
Blau: <INPUT type="radio" value="blau" name="Farbe">
Grün: <INPUT type="radio" value="gruen" name="Farbe">
</FORM>
<A href="javascript:Linkfarbe()">Linkfarbe &auml;ndern</A>
</BODY>
</HTML>
```

Listing 6.13: Drei Farben stehen für die Hyperlinks zur Auswahl.

Innerhalb der gezeigten Syntax befindet sich ein Formular, in welchem wiederum drei Radiobuttons definiert wurden. Unterhalb des Formulars ist ein Hyperlink integriert, bei dessen Anklicken die Funktion `Linkfarbe()` aufgerufen wird. Diese Funktion überprüft, welcher der Radiobuttons vom Anwender selektiert wurde, und weist dem Hyperlink hieraufhin den entsprechenden Farbwert zu.

6.13 Herkunft

unterschiedliche Behandlung in den Browsern

Durch die **referrer**-Eigenschaft kann der URL der Datei ausgelesen werden, von welcher die aktuell angezeigte Datei aufgerufen wurde. Sinnvoll ist dies vor allem im Hinblick auf das Aussperren von Personen, die Ihre Seiten von einem für Sie ungeliebten URL aufgerufen haben. Probleme gibt es hinsichtlich der unterschiedlichen Browserinterpretation. Der Internet Explorer gibt nur dann einen Wert zurück, wenn die aktuelle Datei über das HTTP-Protokoll aufgerufen wurde. Ebenso verhält sich auch Netscapes Browser ab der Produktversion 6. Ältere Netscape-Produkte liefern aber auch dann einen Wert zurück, wenn die entsprechende Datei nicht über das HTTP-Protokoll aufgerufen wurde.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
```

```
function Ursprung()
{
if(document.referrer != "index.html")
location.href = "content.htm";
else
location.href = "sonicht.htm";
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="Ursprung()">
</BODY>
</HTML>
```

Listing 6.14: Eine Seite wird in Abhängigkeit von ihrer Herkunft geladen.

Die gezeigte Syntax nimmt an, dass innerhalb eines Internetprojekts Informationen vorhanden sind, die nur dann ordnungsgemäß zu erreichen sind, wenn der Anwender nicht über die Datei `index.html` darauf zugreift. In diesem Fall, also wenn der **`referrer`**-Wert ungleich `index.html` ist, wird an die Datei `content.htm` weitergeleitet. Anderenfalls gelangt der Anwender auf die Datei `sonicht.htm`, also die korrekte Seite.

6.14 Datei-Titel

Dateititel in jedem Fall angeben

Der Titel einer HTML-Datei sollte im Normalfall zur „Grundausstattung" gehören. Zum einen lässt sich hierdurch eine für den Anwender angenehmere Übersicht realisieren, aber auch im Hinblick auf JavaScript-Funktionen hat der Dateititel durchaus seinen Reiz. Die **`title`**-Eigenschaft ermöglicht den direkten Zugriff auf den Dateititel, wie dieser innerhalb des **`<TITLE>`**-Tags im Dateikopf angegeben wurde. Ein Beispiel:

```
<HTML>
<HEAD>
<TITLE>Produkte</TITLE>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
 document.write("<P>Werter Kunde</P>");
 document.write("Der Titel der aktuellen Seite ist: " +  document.title);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.15: Der Datei-Titel wird in das Dokument geschrieben.

Die gezeigte Syntax hat zur Folge, dass innerhalb der aktuellen Datei dynamisch der Titel `Produkte` geschrieben wird. Sinnvoll ist eine solche Anwendung vor allem, wenn man sich vor Augen führt, dass sich die gezeigte Datei innerhalb eines Framesets befindet und somit deren Titel für den Anwender zunächst nicht sichtbar ist.

6.15 URL der Datei

Auch Änderungen des URLs sind möglich!

Durch das **`location`**-Objekt ist es möglich, den gesamten URL eines Fensters zu speichern. Praktische Anwendungen ergeben sich hieraus allerdings erst im Zusammenhang mit einer geeigneten Funktion. Um einen URL zu speichern und anschließend auf diesen zugreifen zu können, steht für das **`location`**-Objekt die Methode **`href`** zur Verfügung. Exemplarisch soll deren Einsatz und eine mögliche Anwendung hier vorgestellt werden. Im folgenden Beispiel wird dem Anwender zunächst ein simpler textbasierter Hyperlink angezeigt. Durch dessen Anklicken wird die Funktion `gehezudemURL( )` ausgelöst. Diese öffnet ein Eingabefenster, in welchem der Nutzer durch einen Hinweis ermutigt wird, eine beliebige Adresse einzugeben. Nach dieser Eingabe wird der notierte URL in der Variablen `URL` gespeichert. Mit der anschließenden Zuweisung von `URL` zur Methode **`href`** wird das Hyperlinkverhalten zum eingegebenen URL ausgeführt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function gehezudemURL()
{
URL = window.prompt("Geben Sie den gewünschten URL an:","");
window.location.href = URL;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:gehezudemURL()">Zu Ihrer Lieblingsseite</A>
</BODY>
</HTML>
```

Listing 6.16: Der im Dialogfeld notierte URL wird aufgerufen.

Um eine solche Anwendung tatsächlich praxistauglich erscheinen zu lassen, sollte ein Hinweistext den Anwender darauf aufmerksam machen, dass, wenn es sich bei der Zieladresse nicht um eine lokale Datei handelt, der vollständige Pfad samt Protokollangabe notiert werden muss. Will der Anwender also beispielsweise das Verzeichnis Yahoo! erreichen, lautet hier die korrekte Angabe *http://www.yahoo.de*.

6.16 Dokument schließen

Unterschiede zum **window***-Objekt*

Die **close()**-Methode ermöglicht das Schließen eines Dokuments. Beachten Sie, dass sich diese **close()**-Methode von der des **window**-Objekts unterscheidet. Wird **window.close()** dafür eingesetzt, um ein neu geöffnetes Browserfenster zu schließen, verhält sich dies bei **document.close()** anders. Die **document.close()**-Methode schließt einen Dokumentinhalt, welcher über **document.open()** geöffnet und dessen Inhalt mit **write()** bzw. **writeln()** beschrieben wurde. Die hier gezeigte Methode ist vor allem hinsichtlich des Generierens dynamischer Inhalte interessant. Wird doch durch deren Einsatz dem Browser mitgeteilt, dass alle notwendigen Informationen enthalten sind.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
document.open();
document.write("Herzlich willkommen");
document.close();
</SCRIPT>
</BODY>
</HTML>
```

Listing 6.17: Öffnen und Schließen des Dokuments

Über **document.open()** wird der Inhalt des aktuellen Dokuments zum Beschreiben geöffnet. Im Anschluss hieran wird über die **document.write()**-Methode dynamisch der Text *Herzlich willkommen* in das Dokument geschrieben. Um dem Browser den Abschluss des neu definierten Dokumentinhalts mitzuteilen, wird die **close()**-Methode des **document**-Objekts notiert.

6.17 Auf Elemente zugreifen

HTML-Elemente und das DOM

Wie eingangs dieses Kapitels bereits erwähnt, sind alle Elemente einer HTML-Seite in einem Elementbaum angeordnet. Um nun auf diese zugreifen zu können, stehen drei verschiedene Möglichkeiten zur Verfügung. Alle drei Varianten werden vom Internet Explorer ab Version 5 und vom Netscape Navigator ab Version 6 interpretiert. Bei allen Zugriffsvarianten handelt es sich um vom W3C vorgeschlagene DOM-Methoden. Sie haben hierdurch die Möglichkeit, nicht nur auf Formularelemente, sondern beispielsweise auch auf das **<P>**-Tag zuzugreifen. Die folgende Übersicht zeigt alle HTML-Elementobjekte, auf die Sie mittels der HTML-Variante des DOM zugreifen können.

Verfügbare HTML-Elemente				
<A>	<CODE>	<HR>	<OBJECT>	<SUP>
<ABBR>	<COL>	<HTML>	<OL>	<TABLE>
<ACRONYM>	<COLGROUP>	<I>	<OPTGROUP>	<TBODY>
<ADDRESS>	<DD>	<IFRAME>	<OPTION>	<TD>
<APPLET>	<DEL>	<IMG>	<P>	<TEXTAREA>
<AREA>	<DFN>	<INPUT>	<PARAM>	<TFOOT>
<B>	<DIR>	<INS>	<PRE>	<TH>
<BASE>	<DIV>	<ISINDEX>	<Q>	<THEAD>
<BASEFONT>	<DL>	<KBD>	<S>	<TITLE>
<BDO>	<DT>	<LABEL>	<SAMP>	<TR>
<BIG>	<EM>	<LEGEND>	<SCRIPT>	<TT>
<BLOCKQUOTE>	<FIELDSET>	<LI>	<SELECT>	<U>
<BODY>	<FONT>	<LINK>	<SMALL>	<UL>
 	<FORM>	<MAP>	<SPAN>	<VAR>
<BUTTON>	<FRAME>	<MENU>	<STRIKE>	
<CAPTION>	<FRAMESET>	<META>	<STRONG>	
<CENTER>	<H1-H6>	<NOFRAMES>	<STYLE>	
<CITE>	<HEAD>	<NOSCRIPT>	<SUB>	

Tabelle 6.1: Verfügbare HTML-Elemente in alphabetischer Reihenfolge

weiterführende Informationen zu Eigenschaften und Methoden

Beachten Sie, dass jedes der aufgeführten Elemente Eigenschaften und Methoden besitzt. Bei den Eigenschaften handelt es sich um die Attribute, die für die entsprechenden Elemente in der HTML-4.0-Spezifikation festgelegt sind. Die jeweiligen Methoden wurden im DOM festgelegt. Weiterführende Informationen hierzu finden Sie auf den Seiten des W3C unter *http://www.w3.org*.

6.17.1 per ID

Verwendung des **id***-Attributs von HTML-Tags*

Über die **getElementById()**-Methode kann auf HTML-Elemente entsprechend der HTML-Variante des DOM zugegriffen werden. Um dies zu realisieren, muss das betreffende HTML-Element durch die Zuweisung des **id**-Attributs als dateiweit eindeutig gekennzeichnet sein. Als Parameter wird der Wert des **id**-Attributs erwartet. Beachten Sie in jedem Fall, dass zwischen Groß- und Kleinschreibung unterschieden wird. Im folgenden Beispiel wird über **getElementById()** auf einen Textabsatz zugegriffen. Der hierin enthaltene Text wird anschließend in einem mehrzeiligen Eingabefeld angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function TextimTextfeld()
{
document.Formular.Ausgabe.value =
document.getElementById("Text").firstChild.data;
```

```
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P id="Text">Ein Text im &uuml;bertragenen Sinn.</P>
<FORM name="Formular">
<TEXTAREA name="Ausgabe"></TEXTAREA>
<INPUT type="button" onclick="TextimTextfeld()" value="go">
</FORM>
</BODY>
</HTML>
```

Listing 6.18: Der Text wird im Eingabefeld angezeigt.

Das **<P>**-Tag wird durch die ID `Text` als dateiweit eindeutig gekennzeichnet. Innerhalb der Funktion `TextimTextfeld()` wird der Wert des mehrzeiligen Eingabefeldes `Ausgabe` mit dem Text des Absatzes gleichgesetzt. Um auf den Inhalt des **<P>**-Tags zugreifen zu können, bedient sich das Beispiel der Anweisung **`firstChild.data`**, wobei **`firstChild`** den Zugriff auf das Kindobjekt des **<P>**-Elements gestattet. Die Eigenschaft **`data`** dient dazu, den Text dieses Elements zu speichern.

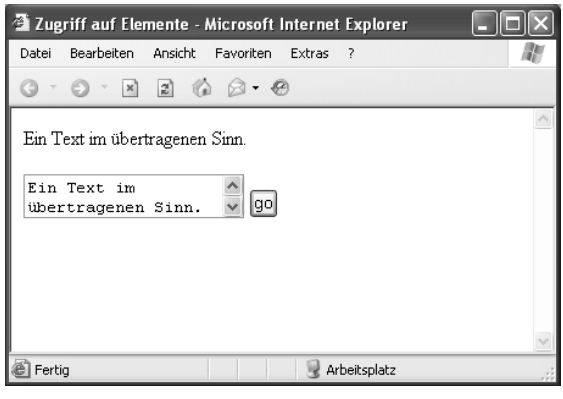


Abbildung 6.4: Der Text wird nun auch im Eingabefeld angezeigt.

6.17.2 Per Name

zwischen Groß- und Kleinschreibung wird unterschieden

Eine weitere Möglichkeit, auf HTML-Elemente zuzugreifen, besteht darin, dass Sie sich des **`name`**-Attributs von HTML-Tags bedienen. Dieses Attribut dient in HTML dazu, ein Element zu kennzeichnen. Durch **`getElementsByName()`** und die Verwendung des Namens eines HTML-Elements können Sie nun auf dieses zugreifen. Als Parameter wird der Wert des **`name`**-Attributs erwartet. Beachten Sie, dass zwischen Groß- und Kleinschreibung unterschieden wird. Der Parameter der **`getElementsBy-`**

Name()-Methode muss also exakt so geschrieben werden wie der Wert des **name**-Attributs im HTML-Tag. Beachten Sie, dass das **name**-Attribut innerhalb einer Datei mehrmals vorkommen darf. Aus diesem Grund muss eine weitere Anweisung hinzugefügt werden. Erst hierdurch kann das betreffende HTML-Element eindeutig angesprochen werden. Schauen Sie sich hierzu die nachfolgende Syntax an. Diese erlaubt, den angezeigten Text des Formular-Buttons nach dessen Anklicken zu verändern. Zunächst wird dem Anwender ein Button mit dem Beschriftungstext `Absenden` angezeigt. Nach dem Anklicken ändert sich dieser Text in `Danke`.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function wandelbarerButton()
{
 document.getElementsByName("Sender")[0].value = "Danke";
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<BUTTON name="Sender" value="" onclick="wandelbarerButton()">Absenden</BUT-
TON>
</FORM>
</BODY>
</HTML>
```

Listing 6.19: Ein wandelbarer Formular-Button

Der Formular-Button ist mit dem **name**-Attribut und dessen Wert `Sender` versehen. Auf den Button kann nun über **getElementsByName("**`Sender`**")** zugegriffen werden. Als weitere Angabe muss in diesem Beispiel noch **[0]** angegeben werden. Dies besagt, dass auf das erste Element mit dem Wert `Sender` des **name**-Attributs zugegriffen werden soll. Würde sich in dem Beispiel ein zweites Element mit dem gleichen Namen befinden, müsste, um auf dieses zugreifen zu können, die **Anweisung getElementsByName("**`Sender`**")[1]** notiert werden.

6.17.3 Per Tag-Name

Unterschiede zwischen der XML- und der HTML-Variante des DOM

Beachten Sie, dass die **getElementsByTagName()**-Methode nicht zur HTML-Variante des DOM gehört. Vielmehr handelt es sich hierbei um die XML-Variante des DOM. Dennoch lässt sie sich auch auf HTML-Elemente anwenden. Dass dies funktioniert, liegt an den Fähigkeiten der Java-

Script-Interpreter moderner WWW-Browser. Als Parameter der **getElementsByTagName()**-Methode wird der Name des gewünschten HTML-Tags erwartet. Soll beispielsweise auf ein **<P>**-Tag zugegriffen werden, muss als Parameter **P** angegeben werden. Für jedes HTML-Element wird innerhalb der Datei ein Array erzeugt. Über dieses kann dann auf das gewünschte Element zugegriffen werden. Soll also auf das zweite **<P>**-Element einer Datei zugegriffen werden, muss die Anweisung **[1]** notiert werden. Im folgenden Beispiel befinden sich u.a. eine sortierte Liste sowie ein einzeiliges Eingabefeld. Der Text des zweiten Listeneintrags wird nach dem Anklicken des Formular-Buttons innerhalb des Eingabefeldes angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
var Zeiger = document.getElementsByTagName("OL")[0].lastChild;
document.Formular.Inhalt.value = Zeiger.lastChild.data;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<OL>
<LI>Herzlich</LI>
<LI>Willkommen</LI>
</OL>
<FORM name="Formular">
<INPUT type="text" name="Inhalt" value="">
<INPUT type="button" value="los" onclick="zeigeWert()">
</FORM>
</BODY>
</HTML>
```

Listing 6.20: Ein Listeneintrag wird im Eingabefeld angezeigt.

Über die Anweisung **getElementsByTagName("OL")[0]** wird auf das erste **<OL>**-Element der Datei zugegriffen. Der zweite Listeneintrag ist der letzte Kindknoten des **<OL>**-Elements und wird über **lastChild** angesprochen. Dieser Wert wird in der Variablen `Zeiger` gespeichert. Anschließend wird der Wert des Eingabefeldes `Inhalt` mit dem Wert `Zeiger`**.lastChild.data** belegt. Diese Anweisung ist notwendig, um auf den Text des Listeneintrags zuzugreifen. Schließlich stellt der Text ein Kindelement des **<LI>**-Elements dar. Der Text muss abschließend noch über **data** ausgelesen werden.

Abbildung 6.5: Der Text aus der Liste wird nun auch im Eingabefeld angezeigt.

6.18 Knoten von Dokumenten

Dokumente bestehen aus Knoten

Bei dem **node**-Objekt handelt es sich um das wichtigste Objekt des DOM. Jedes HTML-Dokument besteht aus einem Elementbaum von Knoten, wobei jedes Element einen eigenen Knoten darstellt. Knoten sind somit beispielsweise HTML-Tags, Attribute und Attribut-Werte. Das **node**-Objekt ist für alle XML-basierten Sprachen, also auch für HTML, verfügbar. Sie können mit dem **node**-Objekt auf alle Elemente einer Datei zugreifen und diese verändern. So lassen sich z.B. HTML-Tags, also Knoten, in das bestehende Dokument dynamisch einfügen. Um das **node**-Objekt verwenden zu können, wird ein Knoten benötigt. Es kann auf bereits bestehende Knoten zugegriffen, es können aber auch neue Knoten erzeugt werden. Für den Zugriff auf bestehende Knoten stehen die im Abschnitt 6.17 vorgestellten Varianten **getElementById**, **getElementsByName** und **getElementsByTagName** zur Verfügung. Wie sich neue Knoten erzeugen lassen, erfahren Sie im Laufe dieses Abschnitts.

Eigenschaften

Eigenschaften von Knoten

- **attributes** – Speichert ein Array aus allen verfügbaren Attributen eines Elements.
- **childNotes** – Speichert ein Array aller vorhandenen Kindknoten eines Knotens.
- **data** – Speichert den Inhalt von Knoten. Dies gilt jedoch nur dann, wenn es sich um einen Textknoten handelt.
- **firstChild** – Speichert das Objekt des ersten Kindknotens eines Knotens.
- **lastChild** – Speichert das Objekt des letzten Kindknotens eines Knotens.
- **nextSibling** – Speichert von einem Knoten den nächsten Knoten im Dokumentbaum.
- **nodeName** – Speichert den Knotennamen.
- **NodeType** – Speichert den Knotentyp.

- **nodeValue** – Speichert den Inhalt oder Wert eines Knotens.
- **parentNode** – Speichert den Elternknoten eines Knotens.
- **previousSibling** – Speichert von einem Knoten den vorhergehenden Knoten im Dokumentbaum.

ein Beispiel zur Einführung

Wie sich die Eigenschaften des **node**-Objekts einsetzen lassen, soll anhand einer Beispielsyntax verdeutlicht werden. In der folgenden Datei befindet sich ein **<DIV>**-Tag mit den beiden Attributen **align** und **onmouseover**. In einem Meldungsfenster soll die Anzahl der Attribute des **<DIV>**-Tags ausgegeben werden. Hierfür bedienen wir uns der **attributes**-Eigenschaft. Beachten Sie, dass dieser Eigenschaft eine weitere Eigenschaft zugewiesen werden muss. Anderenfalls liefert sie nur ein Objekt zurück. Da die **attributes**-Eigenschaft ein Array speichert, können wir uns der Array-Eigenschaft **length** bedienen. Hierdurch kann die Anzahl der vorhandenen Attribute als numerischer Wert gespeichert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeAnzahl()
{
alert(document.getElementsByTagName('DIV')[0].attributes.length + " Attribute
sind enthalten");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<DIV align="enter" onmouseover="zeigeAnzahl()">Textinhalt</DIV>
</BODY>
</HTML>
```

Listing 6.21: Die Anzahl der Attribute wird ausgegeben.

Innerhalb der `zeigeAnzahl()`-Funktion wird über **getElementsByTagName()** auf das **<DIV>**-Element zugegriffen. Durch die Anweisung **attributes.length** wird die Anzahl der vorhandenen Attribute gespeichert. Der ermittelte Wert wird über die **alert()**-Methode ausgegeben. Während der Netscape Navigator 6 den korrekten Wert, nämlich `2`, ausgibt, wird im Internet Explorer der Wert `82` ausgegeben.

Methoden von Knoten

Methoden

- **appendChild()** – Fügt einem Knoten einen Kindknoten hinzu. Als Parameter wird der Elternknoten erwartet.
- **appendData()** – Fügt einem Textknoten Daten am Ende hinzu. Als Parameter wird der Textknoten erwartet.

- **cloneNode()** – Kopiert einen Knoten. Erwartet als Parameter **true**, wenn Element und Inhalt kopiert werden sollen, oder **false**, wenn nur das Element kopiert werden soll.
- **deleteData()** – Löscht die Daten eines Knotens. Als Parameter werden das erste zu löschende Zeichen (0 für das erste) und die Anzahl der zu löschenden Zeichen erwartet.
- **getAttribute()** – Ermittelt den Wert eines Attributs. Als Parameter wird das Attribut erwartet.
- **getAttributeNode()** – Ermittelt den Attributknoten. Als Parameter wird das Attribut erwartet.
- **hasChildNodes()** – Ermittelt, ob ein Knoten einen Kindknoten besitzt. Erwartet keinen Parameter. Als Rückgabewerte werden **true**, wenn das Element Kindknoten besitzt, oder **false**, wenn das Element keine Kindknoten besitzt, geliefert.
- **insertBefore()** – Fügt innerhalb eines Knotens einen Kindknoten vor einem anderen Kindknoten ein. Als Parameter wird der Name des Knotens erwartet.
- **insertData()** – Fügt Daten in einen Textknoten ein. Als Parameter werden die Stelle des Zeichens, ab dem eingefügt werden soll, sowie der einzufügende Text erwartet.
- **removeAttribute()** – Löscht den Attributwert eines Elements. Als Parameter wird das zu löschende Attribut erwartet.
- **removeAttributeNode()** – Löscht das Attribut sowie den Wert eines Elements. Als Parameter wird das Objekt des Attributknotens erwartet.
- **removeChild()** – Löscht einen Kindknoten. Als Parameter wird der zu löschende Kindknoten erwartet.
- **replaceChild()** – Ersetzt einen Kindknoten durch einen anderen. Als Parameter werden der neue Knoten und der zu ersetzende Knoten erwartet.
- **replaceData()** – Ersetzt Zeichendaten eines Elements oder den Wert eines Attributs. Als Parameter werden die Startposition, ab der gelöscht werden soll (0 für das erste Zeichen), sowie der neue Inhalt erwartet.
- **setAttribute()** – Weist einem Attribut einen Wert zu. Als Parameter werden der Name des Attributs und dessen Wert erwartet.
- **setAttributeNode()** – Weist einem Element ein Attribut zu. Als Parameter wird das Attribut erwartet.

Wie sich die Methoden des **node**-Objekts nutzen lassen, wird anhand des folgenden Beispiels gezeigt. Hierin soll ein vorhandener Textknoten gelöscht werden. Bei dem Textknoten handelt es sich um den Inhalt des **<P>**-Tags.

Abbildung 6.6: Die Seite direkt nach dem Laden

Die verwendete Methode lautet **deleteData**. Beim Laden der Seite werden ein Textabsatz sowie ein Hyperlink angezeigt. Das Anklicken des Verweises löscht den Inhalt des Textabsatzes. Dem Anwender wird anschließend nur noch der Hyperlink angezeigt. Beachten Sie, dass nicht das **<P>**-Element, sondern ausschließlich dessen Inhalt gelöscht wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function loescheText()
{
var Anzahl = document.getElementsByTagName("p")[0].
firstChild.nodeValue.length;
document.getElementsByTagName("P")[0].
firstChild.deleteData(0,Anzahl);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P>Guten Tag!
<BR>
<A href="javascript:loescheText()">Text entfernen</A></P>
</BODY>
</HTML>
```

Listing 6.22: Der Text wird entfernt.

Abbildung 6.7: Der Knoten, und somit der Text, ist gelöscht.

Innerhalb der `loescheText()`-Funktion wird zunächst die Variable `Anzahl` deklariert. Diese greift auf den Zeicheninhalt des **<P>**-Elements zu. Da der Inhalt ein Kindelement von **<P>** ist, wird die **firstChild**-Eigenschaft verwendet. Über die Anweisung **nodeValue.length** wird die Anzahl der vorhandenen Zeichen ermittelt. Anschließend wird über **document.getElementsByTagName ("P")[0].firstChild** auf den Inhalt des Textabsatzes zugegriffen. Um den gesamten Text zu löschen, wird die Anweisung **deleteData(0,**`Anzahl`**)** verwendet. Diese legt fest, dass ab dem ersten Zeichen gelöscht werden soll. Die Anzahl der zu löschenden Zeichen wurde zuvor in der Variablen `Anzahl` gespeichert. Da hier die Gesamtzahl der vorhandenen Zeichen gespeichert ist, werden alle Textzeichen gelöscht.

6.18.1 Attribute für den Elementbaum erzeugen

Elemente detailliert beschreiben

Attribute dienen der genaueren Beschreibung eines Elements. Zusätzlich hierzu muss einem Attribut ein Wert zugewiesen werden. Als Beispiel dafür sei eine HTML-Tabelle genannt. Durch die Zuweisung der **border="5"**-Anweisung im einleitenden **<TABLE>**-Tag wird der Browser dazu veranlasst, die Tabelle mit einer Rahmenstärke von **5** Pixel darzustellen. Hierbei ist **border** ein Attribut des **<TABLE>**-Tags. Die Wertzuweisung des Attributs erfolgt durch ein Gleichheitszeichen und den sich hieran anschließenden Wert. In dem genannten Beispiel ist der Wert des **border**-Attributs demnach **5**. Über die **createAttribute()**-Methode lässt sich für den Elementbaum ein neues Attribut erzeugen. Im folgenden Beispiel wird eine Tabelle angezeigt. Diese wird ohne Rahmen dargestellt. Durch die Verwendung der **createAttribute()**-Methode wird dem **<TABLE>**-Tag das **border**-Attribut zugewiesen. Um den Wert des neu erstellten Attributs festzulegen, wird die **nodeValue**-Eigenschaft verwendet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Tabellenrahmen()
{
var Rahmen = document.createAttribute("border");
Rahmen.nodeValue = "5";
var erzeugen = document.getElementsByTagName("TABLE")[0];
erzeugen.setAttributeNode(Rahmen);
}
-->
</SCRIPT>
</HEAD>
<BODY>
<TABLE>
<TR>
<TD>Inhalt</TD>
</TR>
</TABLE>
<A href="javascript:Tabellenrahmen()">Rahmen</A>
</BODY>
</HTML>
```

Listing 6.23: Der Tabelle wird ein fünf Pixel breiter Rahmen zugewiesen.

Die Funktion `Tabellenrahmen()` wird durch Anklicken des Hyperlinks ausgelöst. Innerhalb der Funktion wird zunächst die Variable `Rahmen` deklariert. Dieser werden das **`document`**-Objekt sowie die Anweisung **`createAttribute("border")`** zugewiesen. Um den Wert des **`border`**-Attributs zu bestimmen, wird der **`nodeValue`**-Eigenschaft der Wert `5` zugewiesen. Hierbei handelt es sich um die letztendliche Rahmenstärke der Tabelle. Diese soll demnach *5* Pixel betragen. Im Anschluss hieran wird die Variable `erzeugen` deklariert. Hier wird über die Methode **`getElementsByTagName()`** auf das erste **`<TABLE>`**-Tag der Datei zugegriffen. Um den Rahmen letztendlich der Tabelle zuweisen zu können, wird die **`setAttributeNode()`**-Methode verwendet. Diese fügt den neu erzeugten Attributknoten in das **`<TABLE>`**-Element ein.

6.18.2 Elemente für den Elementbaum erzeugen

neue HTML-Tags mit Inhalten einfügen

Sie können in ein HTML-Dokument ein neues Element einfügen. So lässt sich beispielsweise an einer beliebigen Stelle eine neue Überschrift integrieren. Um dem Elementbaum ein neues Element zuzufügen, wird die **`createElement()`**-Methode verwendet. Als Parameter wird das zu erstellende Element erwartet. Soll zum Beispiel ein neues **`<I>`**-Element ein-

gefügt werden, muss die Anweisung **createElement("i")** notiert werden. Im folgenden Beispiel werden zunächst lediglich ein einzeiliges Eingabefeld sowie ein Formular-Button angezeigt. Das Besondere an dieser Syntax ist, dass der in das Eingabefeld eingetragene Text dynamisch oberhalb des Eingabelfeldes in kursiver Schrift angezeigt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Texteinfuegen()
{
 var Text = document.inhalt.Eingabe.value;
 var kusriv = document.createElement("i");
 var meinText = document.createTextNode(Text);
 kusriv.appendChild(meinText);
 var einfuegen = document.getElementById("Absatz");
 einfuegen.appendChild(kusriv);
}
//-->
</SCRIPT
</HEAD>
<BODY>
<P id="Absatz"></P>
<FORM name="inhalt">
<INPUT type="text" name="Eingabe" value"">
<INPUT type="button" value="Text" onclick="Texteinfuegen()">
</FORM>
</BODY>
</HTML>
```

Listing 6.24: Der eingetragene Text wird im Dokument angezeigt.

Innerhalb der `Texteinfuegen()`-Funktion wird zunächst die Variable `Text` deklariert. Als Wert wird dieser der Wert des Eingabefeldes `Eingabe` zugewiesen. Die Variable `kursiv` erhält den Wert **createElement("I")**. Diese Anweisung bedeutet, dass ein neues **<I>**-Tag erzeugt werden soll. Jetzt kann zwar das Element bereits erstellt werden, Inhalt besitzt es jedoch noch nicht. Hierzu muss über die Anweisung **createTextNode()** ein neuer Textknoten erzeugt werden. Der Inhalt dieses Textes ergibt sich aus dem Wert der Variablen `Text`, also dem Wert des Eingabefeldes. Über **appendChild(meinText)** wird der Text als Kindelement des **<I>**-Elements gekennzeichnet. Um das gesamte **<I>**-Element in den Elementbaum einzufügen, wird über **getElementById("**`Absatz`**")** auf das bereits vorhandene **<P>**-Element zugegriffen. Durch die Anweisung `einfuegen`**.appendChild(**`kusriv`**)** wird das erzeugte **<I>**- dem vorhandenen **<P>**-Element als Kindelement zugewiesen.

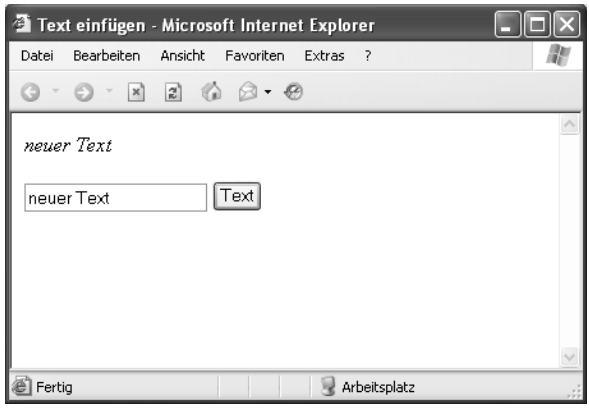


Abbildung 6.8: Der Text aus dem Eingabefeld wird nun auch über diesem angezeigt.

6.18.3 Textknoten für den Elementbaum erzeugen

neuen Text einfügen

Durch die Verwendung der **`createTextNode()`**-Methode kann ein neuer Textknoten innerhalb des Elementbaums erzeugt werden. Sie haben hierdurch die Möglichkeit, an einer beliebigen Stelle des Dokuments Text einzufügen. Beachten Sie, dass die Erzeugung eines Textknotens noch nicht dessen Anzeige bedeutet. Hierzu müssen weitere Elemente verwendet werden. Es muss bestimmt werden, wo und wie der Textknoten angezeigt werden soll. Hierzu ein Beispiel. Beim Laden der Seite wird zunächst lediglich ein Hyperlink angezeigt. Wird dieser angeklickt, erscheint oberhalb dieses Verweises der Text `Ein neuer Text`. Dargestellt wird dieser mittels des **`<H1>`**-Tags, also als Überschrift der ersten Ordnung.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Texteinfuegen()
{
 var Text = document.createTextNode("Ein neuer Text");
 document.getElementsByTagName("H1")[0].appendChild(Text);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<H1></H1>
<A href="javascript:Texteinfuegen()">Text einf&uuml;gen</A>
</BODY>
</HTML>
```

Listing 6.25: Eine neue Überschrift wird oberhalb des Eingabefeldes angezeigt.

Der Variablen Text wird als Wert die Anweisung **createTextNode("**Ein neuer Text**")** zugewiesen. Hierdurch wird ein neuer Textknoten erzeugt. Um diesen anzuzeigen, muss bestimmt werden, welches Element hierfür verwendet werden soll. Innerhalb des Dateikörpers wurde hierfür das **<H1>**-Tag reserviert. Auf dieses wird über **getElementsByTagName()** zugegriffen. Über die Anweisung **appendChild(**Text**)** wird der Inhalt der Variablen Text dem **<H1>**-Element als Kindknoten zugewiesen.

6.19 Selektierten Text anzeigen

unterschiedliche Browser-Interpretationen

Im Netscape Navigator kann durch den Einsatz der **getSelection()**-Methode vom Anwender markierter Text ermittelt werden. Beachten Sie, dass diese Methode im Internet Explorer nicht zur Verfügung steht. Im folgenden Beispiel werden dem Anwender u.a. ein Willkommenstext sowie ein einzeiliges Eingabefeld angezeigt. Selektiert der Anwender den ganzen bzw. einen Bereich des Textes, wird der selektierte Text innerhalb des Eingabefeldes angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Ausgabe()
{
var Zeiger = document.getSelection();
document.Auswahl.Markierer.value = Zeiger;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P>Herzlich Willkommen</P>
<FORM name="Auswahl">
<INPUT type="text" name="Markierer" value="">
<INPUT type="button" value="Text anzeigen" onclick="Ausgabe();">
</FORM>
</BODY>
</HTML>
```

Listing 6.26: Der selektierte Text wird im Eingabefeld angezeigt.

Die Funktion Ausgabe() wird durch Anklicken des Formular-Buttons aufgerufen. Der Variablen Zeiger wird die Anweisung **document.getSelection()** zugewiesen. Im nächsten Schritt wird der Wert des Eingabefeldes mit dem selektierten Text gleichgesetzt. Hieraus resultiert, dass der markierte Text als sichtbarer Wert des Eingabefeldes Markierer angezeigt wird.

Abbildung 6.9: Der selektierte Text wird im Eingabefeld angezeigt.

Der gleiche Effekt lässt sich auch für den Internet Explorer realisieren. Die Syntax muss hierfür allerdings leicht modifiziert werden. Nachfolgend die veränderte Funktion, die in dieser Form das gleiche Ergebnis wie im Netscape Navigator auch für den Internet Explorer ermöglicht.

```
function Ausgabe()
{
var Zeiger = document.selection.createRange().text;
document.Auswahl.Markierer.value = Zeiger;
}
```

Zwar kennt der Internet Explorer die **`getSelection()`**-Methode nicht, dafür aber das **`selection`**-Objekt. Um den selektierten Text zu ermitteln, wird hier die Anweisung **`createRange().text`** verwendet.

6.20 Fragen und Übungen

1. Worin besteht u.a. der Unterschied zwischen dem **`document`**- und dem **`window`**-Objekt?

2. Ändern Sie die Syntax aus Abschnitt 6.5 so um, dass nicht die Farbe der aktiven Hyperlinks, sondern der Hintergrund des Dokuments geändert wird.

7 Frames

lernen

Ziele dieses Kapitels

Durch Frames wird das Anzeigefenster in unterschiedliche Rahmen aufgeteilt. Per JavaScript und über das **frames**-Objekt können Sie alle Frames eines Framesets ansprechen. Von WWW-Browsern werden Frames als eigene Fenster behandelt. Dem Browser ist es also zunächst egal, ob es sich um einen Frame oder ein normales Fenster handelt. Hieraus ergibt sich, dass es sich bei dem **frames**-Objekt um eine nur leicht abgewandelte Form des **window**-Objekts handelt. Alle Eigenschaften und Methoden von **window** können auch auf das **frames**-Objekt angewandt werden. Das **frames**-Objekt kennt lediglich eine Eigenschaft mehr. Zusätzlich zu allen **window**-Eigenschaften kann beim **frames**-Objekt die **length**-Eigenschaft eingesetzt werden. Hiermit kann die Anzahl aller sich innerhalb eines Framesets befindenden Frames ermittelt werden. Später wird im Laufe dieses Abschnitts darauf noch ausführlicher eingegangen.

7.1 Zugriff auf Frames

Eigenschaften und Methoden des **window***-Objekts nutzen*

Obwohl sich auf das **frames**-Objekt alle Eigenschaften und Methoden des **window**-Objekts anwenden lassen, existiert ein gravierender Unterschied. Dieser betrifft die Adressierung der Fenster bzw. Frames. Auf Fenster, deren Eigenschaften und Methoden können Sie bekanntlich über **window.**`Eigenschaft` und **window.**`Methode()` zugreifen. Im Zusammenhang mit Frames stellen sich diese Adressierung und der Zugriff auf Eigenschaften und Methoden folgendermaßen dar. Hinter **parent** bzw. **top** schließt sich das **frames**-Schlüsselwort an. In eckigen Klammern wird eine Indexnummer erwartet. Durch diese wird festgelegt, auf welches Frame zugegriffen werden soll.

```
parent.frames[#].Eigenschaft
parent.frames[#].Methode()
```

Bei dem Schlüsselwort **parent** handelt es sich um einen reservierten Fensternamen. Hiermit wird das Elternfenster des aktuellen Frames an-

gesprochen. An Stelle von **parent** können Sie das ebenfalls reservierte Schlüsselwort **top** verwenden. Durch **top** wird in jedem Fall das oberste Anzeigefenster des Browsers angesprochen. Im nächsten Schritt wird festgelegt, welcher Frame angesprochen werden soll. Hierzu wird das reservierte Wort **frames** mit anschließenden eckigen Klammern verwendet. Innerhalb der Klammern wird eine Indexnummer, mit welcher der gewünschte Frame angesprochen wird, notiert. Die Indexnummern beziehen sich auf die Frameset-Datei. In der Reihenfolge, wie die Frames hierin notiert sind, können diese angesprochen werden. Um auf den ersten Frame dieser Datei zuzugreifen, wird beispielsweise die Anweisung **frames[0]** verwendet. Der zweite Frame kann somit über **frames[1]**, das dritte über **frames[2]** usw. angesprochen werden. Abschließend wird die gewünschte Eigenschaft bzw. Methode notiert.

Frames adressieren

Die Adressierung von Frames durch Indexnummern birgt Gefahren in sich. So führt das Einfügen eines weiteren Frames dazu, dass die Adressierungen häufig nicht mehr stimmen. In einem solchen Fall müssen alle JavaScript-Programme angepasst werden. Um diesem Fall vorzubeugen, können Sie für die Adressierung den Frame-Namen verwenden. Dieser Name wird innerhalb der Frameset-Datei definiert. Hierzu wird im einleitenden **<FRAME>**-Tag dem **name**-Attribut ein entsprechender Frame-Name zugewiesen. Beachten Sie hierbei, dass dieser keine Umlaute oder Sonderzeichen enthalten sollte. Zudem ist bei der Wahl des Frame-Namens darauf zu achten, dass dieser stimmig ist. Als guter Programmierstil hat sich eine geographische Bezeichnung durchgesetzt. So könnte dem rechten Frame eines Framesets beispielsweise der Name `rechtesFrame` zugewiesen werden. Die folgende Syntax gilt für die Verwendung von Frame-Namen.

```
parent.Framename.Eigenschaft
parent.Framename.Methode()
```

Zunächst wird wieder eines der beiden Schlüsselwörter **parent** oder **top** notiert. Hieran schließt sich der Frame-Name an. Achten Sie hier in jedem Fall auf die Unterscheidung zwischen Groß- und Kleinschreibung. Wurde in der Frameset-Datei der Name `rechtesframe` und innerhalb des JavaScripts der Name `rechtesFrame` notiert, führte dies zu einer Fehlermeldung. Hinter dem Frame-Namen folgt die Notation der gewünschten Eigenschaft bzw. Methode.

Unterobjekte

Auf das **frames**-Objekt lassen sich nicht nur die Eigenschaften und Methoden des **window**-Objekts anwenden. Auch alle anderen Objekte, die sich innerhalb der Objekthierarchie unterhalb des **window**-Objekts befinden, können eingesetzt werden. So lässt sich die Syntax **window.location.href** auf Frames folgendermaßen anwenden:

```
parent.rechtesFrame.location.href = "mysql.php";
```

Vor dem gewünschten Objekt und deren Eigenschaft bzw. Methode muss an Stelle des **window**-Objekts die Adressierung des Frames vorgenommen werden. Auf die gleiche Art können alle anderen Objekte auch auf das **frames**-Objekt angewandt werden.

7.2 JavaScripts in anderen Frames nutzen

Funktionen aufrufen

Sie können auf JavaScript-Programme in anderen Frames zugreifen. So ist es beispielsweise möglich, Funktionen aufzurufen oder Variablen-Werte zu manipulieren. Auch hierbei spielt die Adressierung von Frames eine entscheidende Rolle. Im folgenden Beispiel wird ein zweiteiliges Frameset definiert. Im unteren Frame werden drei Checkboxen angezeigt. Durch diese kann der Anwender die Textfarbe auf dieser Seite wählen. Zur Auswahl stehen die Farben Rot, Blau und Grün. Der Funktionsaufruf erfolgt jedoch nicht aus dem unteren, sondern dem oberen Frame. Um diese Anwendung zu realisieren, werden drei Dateien benötigt. Zunächst muss die Frameset-Datei definiert werden. Deren Inhalt stellt sich folgendermaßen dar:

```
<FRAMESET  rows="50%,*">
    <FRAME name="oberesFrame" src="oben.htm">
    <FRAME name="unteresFrame" src="unten.htm">
</FRAMESET>
```

Hierdurch wird das Anzeigefenster in zwei Frames geteilt. Die Teilung findet horizontal statt. Im oberen Frame wird die Datei `oben.htm` angezeigt. Hierin werden dem Anwender die drei Checkboxen zur Farbauswahl präsentiert. Die Funktion `Farbwechsel()` überprüft, welche der Checkboxen selektiert wurde. Hierfür werden drei **if**-Abfragen verwendet. Ist bei einer Checkbox **checked == true** erfüllt, wird über **document.fgColor** auf die Textfarbe des Dokuments zugegriffen und dieser ein entsprechender Farbwert zugewiesen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Farbwechsel()
{
 if(document.the_form.farbe[0].checked == true)
 document.fgColor = "red";
 else if(document.the_form.farbe[1].checked == true)
 document.fgColor = "blue";
 else if(document.the_form.farbe[2].checked == true)
 document.fgColor = "green";
```

```
 else document.alinkColor = "000000";
}
</SCRIPT>
</HEAD>
<BODY>
<p>Was m&ouml;gen Sie am liebsten?</p>
<FORM name="the_form">
<INPUT type="radio" value="rot" name="farbe">
<INPUT type="radio" value="blau" name="farbe">
<INPUT type="radio" value="gruen" name="farbe">
</FORM>
</BODY>
</HTML>
```

Listing 7.1: Drei Radio-Buttons zur Farbauswahl

Um die Funktion `Farbwechsel()` aus einer anderen Datei heraus aufzurufen, muss der normale Funktionsaufruf **`javascript:`**`Farbwechsel()` modifiziert werden.

Wie sich dies realisieren lässt, wird anhand des Inhalts der Datei `unten.htm` gezeigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:parent.oberesFrame.Farbwechsel()">Textfarbe &auml;ndern
</A>
</BODY>
</HTML>
```

Listing 7.2: Ein Frame-übergreifender Funktionsauruf

Um einen Frame-übergreifenden Funktionsaufruf zu programmieren, muss zunächst das Schlüsselwort **`parent`** notiert werden. Im nächsten Schritt wird das Frame, in welchem sich die Funktion befindet, adressiert. Hierbei stehen die beiden vorgestellten Varianten der Adressierung durch Indexnummern sowie der Adressierung durch Frame-Namen zur Verfügung. In dem gezeigten Beispiel wird Letztere genutzt. Es wird hier also über `oberesFrame` auf den Inhalt der Datei `oben.htm` zugegriffen. Abschließend muss noch der Funktionsname, in diesem Beispiel `Farbwechsel()`, notiert werden. Neben Funktionsaufrufen können beispielsweise auch Variablen ausgegeben oder manipuliert werden. Sie können also alle Möglichkeiten von JavaScript-Funktionen auch Frame-übergreifend nutzen.

7.3 Anzahl ermitteln

untergeordnete Frames

Im Laufe dieses Abschnitts wird Ihnen noch des Öfteren die **length**-Eigenschaft des **frames**-Objekts begegnen. Mit dieser Eigenschaft lässt sich die Anzahl von Frame-Fenstern, die ein übergeordnetes Fenster besitzt, ermitteln. Als Rückgabewert wird eine Ziffer geliefert. Enthält das übergeordnete Fenster beispielsweise zwei Frames, wird der Wert *2* gespeichert. Im folgenden Beispiel werden die Namen aller vorhandenen Frames in einem Meldungsfenster ausgegeben. Es wird angenommen, dass sich die aufgeführte Datei innerhalb eines Framesets befindet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
for(var i=0; i < top.frames.length; i++)
alert(top.frames[i].name);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 7.3: Die Namen aller Frames werden nacheinander ausgegeben.

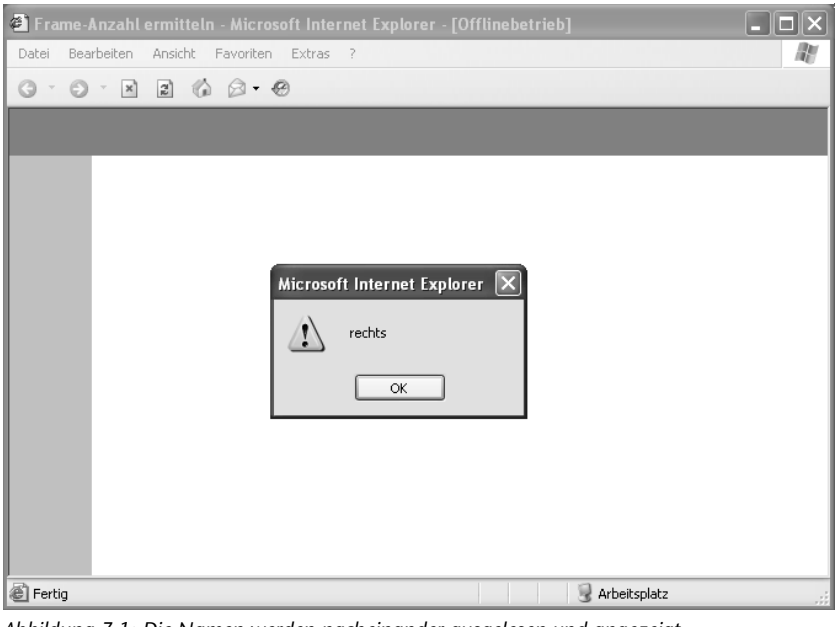


Abbildung 7.1: Die Namen werden nacheinander ausgelesen und angezeigt.

Um alle Namen auszugeben, wird eine **for**-Schleife verwendet. Da es eine Endlosschleife zu verhindern gilt, wird die **length**-Eigenschaft als Abbruchkriterium verwendet. Die Schleife wird so oft durchlaufen, bis der Wert der Variablen `Inhalt` kleiner als der Wert der Eigenschaft **length** ist. Nach jedem Schleifendurchlauf wird der Wert von `Inhalt` um eins erhöht. Um die jeweiligen Frames anzusprechen, wird die Anweisung **frames[**`Inhalt`**]** verwendet. Da es sich bei dem Wert von `Inhalt` um eine Zahl handelt und der Startwert mit `0` definiert wurde, werden alle Frames durchlaufen. Die Frame-Namen werden über die **name**-Eigenschaft ermittelt und über die **alert()**-Methode ausgegeben.

7.4 Frames drucken

unterschiedliche Druckdialoge in den Browsern

Frames sind sicherlich Streitobjekte, bieten aber dennoch einige Vorteile. Sicherlich kein Vorteil ist das Ausdrucken von Frames. Je nach verwendetem Browser muss der Anwender erst einige Einstellungen vornehmen, um ein bestimmtes Frame aus einem Frameset drucken zu können. Sonderlich komfortabel ist dies freilich nicht. Das folgende Script hilft dabei, das Ausdrucken von Frames einfacher zu gestalten. Um das Script zu verstehen, muss zunächst geklärt werden, wie Browser auf Frames reagieren. Vorerst betrachtet der Browser jeden Frame als eigenes Fenster. Soll ein Frame ausgedruckt werden, muss der Anwender entscheiden, welche Frame ausgedruckt werden soll. Hierfür bieten die gängigsten Browser im Druck-Dialog solche Punkte wie „Nur den markierten Frame drucken“ an. Wird diese Option gewählt, wird tatsächlich der Frame ausgedruckt, in dem der Anwender zuletzt eine Aktion, beispielsweise einen Mausklick, ausgeführt hat. Das folgende Beispiel nutzt dieses Prinzip. Es wird angenommen, dass sich diese Seite innerhalb eines Framesets befindet. Im rechten Frame wird ein Hyperlink angezeigt. Dessen Anklicken führt dazu, dass der linke Frame markiert wird und somit relativ problemlos ausgedruckt werden kann.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function druckFrame()
{
   parent.frames.links.focus();
   parent.frames.links.print();
}
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:druckFrame()">drucken</A>
```

```
</BODY>
</HTML>
```

Listing 7.4: Das Frame `links` wird gedruckt.

Innerhalb der Funktion `druckFrame()` werden die beiden Methoden **`focus()`** und **`print()`** verwendet. Um auf den linken Frame zuzugreifen, wird dieser über **`parent.frames.`**`links` angesprochen. Die **`focus()`**-Methode dient dazu, den linke Frame zu markieren. Um den Frame auszudrucken, wird die **`print()`**-Methode eingesetzt. Beachten Sie, dass durch dieses Script nicht der Druck-Dialog des Browsers unterdrückt wird.

7.5 Frameanzeige verhindern

eingezwängte Darstellung umgehen

Häufig kommt es vor, dass Internetseiten in fremden Frames angezeigt werden. Davor gilt es, sich zu schützen! Denn nicht nur werden Ihre Seiten als Teil eines anderen Projekts dargestellt, häufig hat dies auch ästhetische Gründe. In vielen Fällen passen die Inhalte optisch nicht in ein anderes Frameset und das Corporate Design wird verfälscht. Um die Anzeige in anderen Frames zu verhindern, bedient sich das folgende Beispiel der **`length`**-Eigenschaft.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
if (top.frames.length != 0)top.location=self.location;
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Der Script-Bereich wird mit dem Einlesen der Datei ausgeführt. Hierin wird mittels einer **`if`**-Abfrage überprüft, ob sich der Frame innerhalb eines anderen befindet. Um dies zu erreichen, wird über **`top`** auf das oberste Anzeigefenster des Browsers zugegriffen. Im nächsten Schritt wird untersucht, ob die Anzahl der hierin enthalten Frames ungleich `0` ist. Diese Aufgabe übernimmt die Anweisung **`frames.length !=`** `0`. Ist diese Bedingung erfüllt, wird über **`top.location`** auf den URI des oberen Anzeigefensters zugegriffen und dieser mit dem URI der von Ihnen erstellen Seite belegt. Hierfür wird die Anweisung **`self.location`** genutzt. Wird Ihre Datei beispielsweise über den URI *http://ww.myhhost.de/content.htm* aufgerufen, wird der URI des oberen Anzeigefensters auf exakt diesen geändert. Die Datei wird demnach nicht mehr in einem Frame angezeigt. Be-

achten Sie, dass dieses Script auch dann funktioniert, wenn Ihre Seiten selbst als Frameset definiert wurden. In diesem Fall muss der gezeigte Quellcode innerhalb der eigentlichen Frameset-Datei notiert werden.

7.6 Frameanzeige erzwingen

unmögliche Navigation

Im Gegensatz zu dem zuvor gezeigten Problem ist es häufig notwendig, dass die eigenen Seiten in einem Frameset angezeigt werden. Dies ist zumeist dann der Fall, wenn Sie als Entwickler Ihre Seiten als Frames definiert haben. Probleme gibt es immer dann, wenn auf Ihre Seiten verwiesen wird. Häufig werden die zu einem Frameset gehörenden Seiten von Suchmaschinen gefunden und vom Anwender direkt aufgerufen. Und hier beginnen die Probleme. Auf Seiten, die für Frames optimiert sind, fehlt zumeist die Navigation. Der Anwender ist somit recht hilflos, wenn er auf weitere Seiten des Projekts zugreifen möchte. In einem solchen Fall ist es ratsam, die Anzeige einer solchen Datei in einem Frameset zu erzwingen. Ein erstes Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
if (top.frames.length==0) location.replace("index.html")
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 7.5: Die Seite ruft die Datei `index.html` auf.

Hier wird mittels einer **`if`**-Abfrage überprüft, ob die Anzahl der Frames im oberen Anzeigefenster gleich `0` ist. Dafür wird durch **`top`** auf das oberste Anzeigefenster zugegriffen. Die Überprüfung, ob sich hier nur ein Frame befindet, wird durch **`frames.length==`**`0` vorgenommen. Ist diese Bedingung erfüllt, wird der History-Eintrag des oberen Anzeigefensters überschrieben. Hierzu wird die **`replace()`**-Methode des **`location`**-Objekts verwendet. Als Parameter wird der URI der dazugehörenden Frameset-Datei übergeben.

eine höflichere Variante

Die zuvor gezeigte Syntax funktioniert zwar, birgt aber einige Nachteile in sich. Gegenüber dem Anwender ist es nicht sonderlich freundlich, ungefragt Einträge in der History zu überschreiben. Um dieses Problem zu umgehen, kann der Anwender darauf hingewiesen werden, dass die

geladene Datei Teil eines Framesets ist. Zusätzlich hierzu wird ein Hyperlink angezeigt, dessen Verweisziel die Frameset-Datei ist.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
if (top.frames.length==0)
document.write("Die Seite gehört zu einem Frameset. Bitte rufen Sie die Seite
<A href='index.html'>Willkommen</A> auf!");
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 7.6: Der Anwender kann selbst entscheiden, ob er Frames wünscht.

In Form einer **if**-Abfrage wird überprüft, ob im oberen Anzeigefenster die Anzahl der Frames gleich `0` ist. Hierzu wird über **top** auf das obere Anzeigefenster zugegriffen und die Anzahl über **frames.length == 0** überprüft. Ist diese Bedingung erfüllt, wird ein hinweisender Text durch den Einsatz der **write()**-Methode dynamisch in das Dokument geschrieben.

7.7 Aus dem Frameset ausbrechen

ein Button für den Ausbruch

Im folgenden Beispiel wird dem Anwender ein Formular-Button angezeigt. Dies geschieht jedoch nur dann, wenn diese Datei innerhalb eines Framesets geladen wird. Durch Anklicken des Formular-Buttons wird erreicht, dass diese Datei nicht mehr als Teil eines Framesets geladen wird. Das Besondere an dieser Syntax ist, dass kein spezieller URI angegeben werden muss. Somit ist dieses Script für alle Seiten universell einsetzbar.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
Inhalt= "<FORM><INPUT type=\"button\" value=\"ohne Frames\"
onclick=\"top.location.href=self.location.href\"></FORM>"
if (top.frames.length!=0) document.write(Inhalt)
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 7.7: Durch Button-Klick wird die Frame-Anzeige verhindert.

Innerhalb des Script-Bereichs wird die Variable **Inhalt** deklariert. Diese besteht aus der Definition eines Formulars und eines Formular-Buttons. Der Formular-Button wird mit dem Event-Hanlder **onclick** versehen. Dieser löst JavaScript-Code aus. Innerhalb dieses Codes wird über das Schlüsselwort **top** auf das obere Anzeigefenster zugegriffen. Dessen aktueller URI, der über **location.href** angesprochen wird, wird mit dem URI der im Frameset geladenen Datei vertauscht. Hierzu muss der URI dieser Datei über **self.location.href** ausgelesen werden.

7.8 Zum Frameset zurückkehren

Frameset nachladen

Eine vergleichbare Anwendung zur vorhergehenden zeigt die hier vorgestellte Syntax. Auch da wird dem Anwender, unter bestimmten Umständen, ein Formular-Button angezeigt. Dieser bestimmte Umstand ist, dass das aktuelle Dokument zwar Teil eines Framesets ist, jedoch nicht in diesem geladen wird. Tritt also dieser Fall ein, kann der Anwender durch das Anklicken des Formular-Buttons das zur der Seite gehörende Frameset nachladen.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
Inhalt= "<FORM><INPUT type=\"button\" value=\"Frameset anzeigen\"
onclick=\"top.location.href='frameset.html'\"></FORM>"
if (top.frames.length==0) document.write(Inhalt)
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 7.8: Die Frame-Anzeige wird erzwungen.

In dem Script-Bereich, der mit dem Laden der Seite ausgeführt wird, wird zunächst die Variable `Inhalt` deklariert. Als Wert wird dieser Variablen HTML-Code zugewiesen. Dieser besteht aus einem Formular, einem Formular-Button, dem Event-Handler **onclick** sowie einer JavaScript-Anweisung. Durch diese Anweisung wird über das Schlüsselwort **top** auf das oberste Anzeigefenster zugegriffen. Hierin wird nun über die Anweisung **location.href =**`'frameset.html'` die Frameset-Datei geladen. Angezeigt wird der Formular-Button nur dann, wenn die Bedingung **top.frames. length==0**, die besagt, dass im obersten Anzeigefenster kein Frame geladen wurde, erfüllt ist. Erst dann wird über die **write()**-Methode der Wert der Variablen `Inhalt` in das Dokument geschrieben.

7.9 Mehrere Frames gleichzeitig ändern

in HTML ein Problem

Sollen zwei Frames gleichzeitig verändert werden, steht man vor einem Problem. Mit HTML könnte man dieses zwar bewältigen, die Lösung wäre allerdings nicht befriedigend. Müsste doch auf eine neue Frameset-Datei verwiesen werden, in der dann die beiden neuen Frames angezeigt werden. Per JavaScript lassen sich zwei Frames einfacher verändern, und dies, ohne dass eine neue Frameset-Datei aufgerufen werden muss. Für unser Beispiel nehmen wir das folgende Frameset an.

```
<HTML>
<HEAD>
</HEAD>
<FRAMESET cols="20%,*">
 <FRAME src="links.htm" name="linkerFrame">
 <FRAMESET rows="50%,*">
   <FRAME src="rechtsoben1.htm" name="rechtsobenFrame">
   <FRAME src="rechtsunten1.htm" name="rechtsuntenFrame">
 </FRAMESET>
</FRAMESET>
</HTML>
```

Listing 7.9: Das Grundgerüst der Frameset-Datei

Der Anzeigebereich wird in drei Frames unterteilt. Im linken Frame befindet sich ein Hyperlink. Dessen Anklicken soll den Inhalt der beiden rechten Frames gleichzeitig ändern.

Es existieren unterschiedliche Varianten, um das gewünschte Ergebnis zu erzielen. Zunächst eine Variante, die allerdings wenig flexibel ist, und nur in begründeten Ausnahmefällen eingesetzt werden sollte.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zweiFramesaendern()
   {
      parent.rechtsobenFrame.location.href="rechtsoben.htm";
      parent.rechtsuntenFrame.location.href="rechtsunten.htm";
   }
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zweiFramesaendern()">&auml;ndern
</A>
```

```
</BODY>
</HTML>
```

Listing 7.10: Zwei Frames werden gleichzeitig verändert.

Innerhalb der Funktion wird über **parent** und die Fensternamen auf die beiden rechten Frames zugegriffen. Deren URI wird über **locaton.href** angesprochen. Als Werte der **href**-Eigenschaft werden die URIs der beiden zu ladenden Dateien angegeben. Das Anklicken des Verweises ändert die beiden rechten Frames. Die Funktion `zweiFramesaendern()` löst somit das beschriebene Problem. Wirklich elegant ist die gezeigte Lösung allerdings nicht. Was passiert, wenn sich in der Datei mehrere Hyperlinks befinden und diese ebenfalls jeweils zwei Frames gleichzeitig ändern sollen? In diesem Fall müssten mehrere Funktionen vom Typ **zweiFramesaendern()** definiert werden. Eine solche Lösung währe schlechter Programmierstil. Besser geeignet ist hier das Arbeiten mit Parametern. Die folgende Syntax zeigt eine im Ergebnis gleiche Funktion. Diese ist allerdings flexibler als die zuvor aufgeführte.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zweiFramesaendern(Pfad1,Frame1,Pfad2,Frame2)
{
  rechtsoben=eval("parent."+Frame1);
  rechtsunten=eval("parent."+Frame2);
  rechtsoben.location.href = Pfad1;
  rechtsunten.location.href = Pfad2;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zweiFramesaendern('rechtsoben.htm','rechtsoben-
Frame','rechtsunten.htm','rechtsuntenFrame')">&auml;ndern</A>
</BODY>
</HTML>
```

Listing 7.11: Frame-Änderung durch die Verwendung von Parametern

Die Funktion **zweiFramesaendern()** verwendet vier Parameter. Den beiden Variablen `rechtsunten` und `rechtsoben` wird jeweils die **eval()**-Funktion zugewiesen. Diese Funktion sorgt dafür, dass die Fensternamen nicht als Zeichenkette, sondern als Objekte behandelt werden. Als Werte der beiden Variablen werden also die Werte `parent.rechtsoben` und `parent.rechtsunten` gespeichert. Die beiden Variablen `rechtsunten` und `rechtsoben` wer-

den nun für den Aufruf der Dateien verwendet. Hierzu wird hinter den beiden Variablen jeweils die Anweisung **location.href** notiert. Welche Dateien aufgerufen werden sollen, wird durch die Parameter im Funktionsaufruf festgelegt. Dazu werden hinter dem Funktionsnamen jeweils der URI der zu ladenden Datei sowie der Name des Ziel-Frames angegeben. Der Aufruf für den rechten obere Frame wird somit über die Anweisung `'rechtsoben.htm','rechtsobenFrame'` realisiert.

eine elegantere Lösung

Nun ist aber auch diese Lösung noch nicht ganz frei von Unstimmigkeiten. Dies liegt jedoch zunächst einmal nicht an der Funktion selbst, sondern an der Behandlung von Frames durch den Browser. Für den Browser werden beim Aufrufen der Funktion `zweiFramesaendern()` zwei Hyperlinks ausgelöst. Beim Anklicken des Zurück-Buttons des Browsers wird in der History jedoch nur um eine Seite zurückgesprungen. Um also tatsächlich wieder zur Ausgangsposition vor dem Funktionsaufruf zu kommen, muss der Zurück-Button zweimal angeklickt werden. Zwar ließe sich dieses Problem verkraften, komfortabel wäre eine solche Lösung freilich nicht. Die folgende Funktion schafft hier Abhilfe.

```
function zurueck(Frame1,Frame2)
{
  rechtsoben=eval("parent."+Frame1);
  rechtsunten=eval("parent."+Frame2);
  rechtsoben.history.back();
  rechtsunten.history.back();
}
```

Der Funktion `zurueck()` werden zwei Parameter übergeben. Die ersten beiden Zeilen sind aus der Funktion `zweiFramesaendern()` bekannt. Die eigentliche Zurück-Funktionalität wird über die Verwendung der beiden Variablen `rechtsoben` und `rechtsunten` im Zusammenspiel mit der **back()**-Methode des **history**-Objekts realisiert. Durch diese Methode wird die zuletzt besucht Seite aufgerufen. Der Hyperlink, der die Funktion `zurueck()` aufruft, stellt sich folgendermaßen dar:

```
<A href="javascript:zurueck
('rechtsobenFrame','rechtsuntenFrame')">zur&uuml;ck</A>
```

Als Parameter werden der Funktion `zurueck()` die beiden Frame-Namen übergeben. Beachten Sie, dass sich auf eine vergleichbare Weise auch ein vorblättern im Browser realisieren lässt. Für diesen Zweck muss an Stelle der **back()**- die **forward()**-Methode verwendet werden.

7.10 Fragen und Übungen

1. Fügen Sie in das Beispiel auf Seite 248 eine neue Funktion ein, mit deren Hilfe nach vorne geblättert werden kann.

2. Welche Eigenschaft, die im `frames`-Objekt einsetzbar ist, kann im `window`-Objekt nicht eingesetzt werden?

8 Browser und Bildschirm

lernen

Ziele dieses Kapitels

In diesem Kapitel lernen Sie, wie Sie wichtige Informationen über den Anwender einholen können. Sie werden hierdurch in die Lage versetzt zu erkennen mit welchem Browser ein Anwender Ihre Seiten aufruft. Ebenso können Sie die Bildschirmgröße ermitteln oder beispielsweise überprüfen, ob ein benötigtes Plug-In auf dem System des Anwenders installiert ist. All diese Informationen können Sie für die Optimierung Ihrer Seiten nutzen. Sie finden in diesem Kapitel beispielsweise Wege, um Hintergrundgrafiken in Abhängigkeit von der vorhandenen Bildschirmgröße zu laden.

8.1 Alles über den Browser

das navigator-Objekt

Über das **navigator**-Objekt können Sie den vom Anwender verwendeten Browser ermitteln. Zusätzliche Informationen, wie beispielsweise die Plattform, auf der der Browser installiert ist, lassen sich ebenfalls einholen. Auf die Eigenschaften und Methoden des **navigator**-Objekts können Sie folgendermaßen zugreifen:

```
navigator.Eigenschaft
navigator.Methode()
```

Hinter dem Objektnamen **navigator** wird die gewünschte Eigenschaft bzw. Methode notiert. Die folgende Syntax zeigt hierfür zwei Beispiele. Beachten Sie, dass es sich bei der **javaEnabled()**-Methode um die einzige beim **navigator**-Objekt vorhandene Methode handelt.

```
navigator.appName
navigator.javaEnebled()
```

Sie sollten das **navigator**-Objekt immer dann einsetzen, wenn Sie JavaScript-Code verwenden, der nicht von allen Browsern interpretiert werden kann. Somit können Sie in Abhängigkeit vom ermittelten Browser nur das JavaScript ausführen, welches auch tatsächlich vom verwendeten Browser verstanden wird.

8.1.1 Spitzname des Browsers

immer Mozilla

Für JavaScript-Entwickler nur bedingt geeignet ist die **AppCodeName**-Eigenschaft. Zwar lässt sich hiermit der Spitzname des Browsers ermitteln, die Ergebnisse, welche die unterschiedlichen Browser liefern, sind aber dennoch nicht befriedigend. Denn sowohl der Internet Explorer wie auch Opera liefern hierbei den gleichen Wert wie der Netscape Navigator, nämlich *Mozilla*. Dass sich aus diesem Grund eine sinnvolle Anwendung dieser Eigenschaft kaum erkennen lässt, liegt auf der Hand. So eignet sich **AppCodeName** zum Beispiel keineswegs für solche Anwendungen wie Browserweichen usw., da ja hierbei der jeweils gleiche Wert zurückgeliefert wird.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
 document.write("Immer das Gleiche: " + navigator.appCodeName);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.1: Der Spitzname des Browsers wird in das Dokument geschrieben.

In dem gezeigten Beispiel wird mit dem Aufrufen der Datei der Spitzname des verwendeten Browsers dynamisch in das Dokument geschrieben. Das Ergebnis ist hier in jedem Fall *Mozilla*.

8.1.2 Browsername

Opera als Ausnahme

Durch den Einsatz der **appName**-Eigenschaft kann der Name des verwendeten Browsers ermittelt werden. Die Rückgabewerte von **appName** sind für den Internet Explorer *Internet Explorer* und für den Netscape Navigator *Netscape*. Der Opera-Browser meldet sich im Normalfall mit *Opera*. Da in diesem Browser der Anwender aber selbst bestimmen kann, mit welchem Wert sich Opera meldet, sind hier die Ergebnisse häufig verfälscht. Wenngleich die **appName** bereits seit einigen Browserversionen von den Großen interpretiert wird, ist deren Einsatz doch ein gewisses Maß an Skepsis entgegenzubringen. Handelt es sich doch hierbei, und dies gilt gerade im Hinblick auf Browserweichen, um ein wenig geeignetes Mittel. Zwar kann der Browsername ermittelt werden, dieser Wert allein ist jedoch nur wenig aussagekräftig. So gibt es bekannterweise gravierende Unterschiede zwischen dem Netscape Navigator 4 und dem Navigator 6.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
document.write("Sie verwenden den Browser:
<H1>" + navigator.appName + "</H1>");
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.2: Der Browsername wird in das Dokument geschrieben.

Durch die gezeigte Syntax wird anhand der **write()**-Methode der Name des verwendeten Browsers dynamisch in das Dokument geschrieben. Sie sollten diese Syntax auf Grund der aufgezählten und nahe liegenden Nachteile also nur in begründeten Ausnahmefällen einsetzen.

Abbildung 8.1: Der Internet Explorer meldet sich.

8.1.3 Browserversion

verfälschte Ergebnisse

Anhand der **appVersion**-Eigenschaft kann die Produktversion des aktuell verwendeten Browsers gespeichert werden. Bedenken Sie vor dem Einsatz dieser Eigenschaft hinsichtlich einer Browserweiche, dass der hier erhaltene Wert sich für diesen Zweck nur bedingt eignet. So liefert der Internet Explorer 6 beispielsweise den Wert 4. Aber auch der Opera-Browser liefert hier nicht in jedem Fall den korrekten Wert. Bei diesem Produkt kann der Anwender selbst entscheiden, als welcher Browser sich Opera ausgeben soll. Aus all diesen Gründen ist die **appVersion** also für einen praxistauglichen Einsatz nur bedingt geeignet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
 alert(navigator.appVersion);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 8.3: Die Browserversion wird in einem Meldungsfenster ausgegeben.

Beim Laden der Seite wird innerhalb eines **alert()**-Fensters die Produktversion des Browsers ausgegeben. Beachten Sie, dass hierbei jedoch weitaus mehr Informationen als die reine Versionsnummer angezeigt werden. So liefert der Internet Explorer beispielsweise den folgenden Wert: 4.0 (compatible; MSIE 6.0; Windows NT 5.1;Q312461). Um aus einer solchen Zeichenkette die reine Produktversion zu ermitteln, sollte die **substring()**-Methode des **string**-Objekts verwendet werden.

8.1.4 Sind Cookies erlaubt?

Daten speichern

Die Verwendung von Cookies spielt im Hinblick auf diverse Anwendungen eine entscheidende Rolle. Mögliche Einsatzformen dieser kleinen Textdateien reichen von der Personalisierung einer Seite bis hin zu Shop-Systemen. In vielen Fällen, als Beispiel sei hier der E-Mail-Dienst von Yahoo! genannt, kommen Anwender ohne die Aktivierung von Cookies nicht an die angeforderten Daten. Durch die **cookieEnabled**-Eigenschaft kann nun überprüft werden, ob der Anwender das Speichern von Cookies gestattet. Einer der boolschen Werte **true** oder **false** wird vom Browser zurückgeliefert.

mögliche Werte

- **true** – Das Setzen von Cookies wird gestattet.
- **false** – Das Setzen von Cookies wird nicht gestattet.

Beachten Sie, dass viele Anwender ihre Sicherheitseinstellungen dahingehend modifiziert haben, dass vor jedem Setzen von Cookies ein Warnhinweis erscheint. Diesen können Sie als Entwickler nicht unterdrücken. Als Rückgabewert wird in diesem Fall **true** zurückgegeben. Das folgende Beispiel nimmt an, dass sich diese Seite innerhalb eines Shop-Systems befindet. Der Anwender soll nur dann auf die Warenkorb-Seite gelangen, wenn in seinem Browser Cookies akzeptiert werden. Anderenfalls soll sich eine Seite mit einem freundlichen Hinweis bezüglich der Tatsache öffnen, dass ohne aktivierte Cookies keine Bestellung möglich ist.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
if(navigator.cookieEnabled == true)
window.location.href = "warenkorb.htm";
else if(navigator.cookieEnabled == false)
window.location.href = "anschalten.htm";
else alert("Sind Ihre Cookies aktiviert?");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.4: Nur wenn Cookies erlaubt sind, wird die Datei `warenkorb.htm` angezeigt.

Die Syntax geht von drei möglichen Situationen aus, die in Form von **`if`**-Bedingungen abgefragt werden. Gestattet der Anwender-Browser Cookies, wird die Seite `warenkorb.htm` geöffnet. Ist dies nicht der Fall, wird automatisch die Seite `anschalten.htm` angezeigt. Für den Fall, dass der verwendete Browser die Eigenschaft **`cookieEnabled`** nicht interpretieren kann, wird innerhalb eines Meldungsfenster sein entsprechender Hinweis ausgegeben.

8.1.5 Sprachversion des Browsers

mehrsprachige Internetprojekte

In Zeiten zunehmender Internationalisierung stellen multilinguale Internetprojekte bei weitem keine Ausnahme mehr dar. Um Anwendern die Möglichkeit zu geben, auf die für ihren Sprachkreis optimierte Seite zu gelangen, gibt es ja bekanntlich die verschiedensten Varianten. Diese reichen von textbasierten Hyperlinks bis hin zu animierten Landesflaggen. Eine andere, wenn auch nicht in jedem Fall optimale, Lösung bietet die **`language`**-Eigenschaft. Diese speichert die beim Anwender eingestellte Sprache des Browsers. Als Rückgabewert werden hierbei vom Browser die typischen zweistelligen Abkürzungen wie beispielsweise *`de`* für Deutschland oder *`fr`* für Frankreich gespeichert. Das folgende Beispiel beschreibt eine mögliche Anwendung der **`language`**-Eigenschaft.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
```

```
if(navigator.language == "en")
window.location.href = "welcome.htm";
else(navigator.language == "de")
window.location.href = "willkommen.htm";
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.5: Es wird eine Seite in Abhängigkeit von der Sprachversion angezeigt.

Direkt mit dem Laden der Seite wird das JavaScript ausgeführt. In diesem wird in Form einer **if**-Abfrage überprüft, ob die Sprache des Anwender-Browsers Englisch, also den Wert `en`, zurückliefert. Ist dies der Fall, wird der Nutzer auf die Seite `welcome.htm` geleitet. Wird indes der Wert `de` ermittelt, verwendet der Nutzer also einen deutschsprachigen Browser, wird die Seite `willkommen.htm` angezeigt.

Bedenken Sie, dass die gezeigte Lösung keineswegs der Weisheit letzter Schluss ist. Es steht in keinem Fall fest, dass Anwender, die einen englischsprachigen Browser verwenden, auch tatsächlich der englischen Sprache mächtig sind. Wer dennoch die **language**-Eigenschaft für eine solche Sprachenweiche einsetzen möchte, sollte in jedem Fall innerhalb der jeweiligen Sprachversion einen Querverweis auf die andere integriert haben.

keine Kompatibilität

Die beiden „großen" Browser sind bezüglich der **language**-Eigenschaft einmal mehr nicht kompatibel. Während der Netscape Navigator die **language**-Eigenschaft korrekt interpretiert, wird im Internet Explorer eine Fehlermeldung ausgegeben. Um auch für den Microsoft-Browser die Sprachversion herauszufinden, muss die **userlanguage**-Eigenschaft eingesetzt werden.

8.1.6 Betriebssystem

Unterschiede in der Darstellung auf Macintosh- und Windows-Systemen

Internetseiten werden auf verschiedenen Betriebssystemen unterschiedlich dargestellt. Dass dies so ist, erfährt wohl am gravierendsten die Macintosh-Gemeinde. Diese muss permanent die leidige Erfahrung machen, dass nur die wenigsten Internetprojekte auf ihr Betriebssystem abgestimmt sind. So ist die Darstellung viel zu kleiner Schriften hier wohl eines der ärgerlichsten Vorkommnisse. Vor allem, wenn man berücksichtigt, mit wie wenig Aufwand sich dies umgehen ließe. So reicht es beispielsweise aus, das beim Anwender laufende Betriebssystem zu ermitteln und anschließend ein entsprechendes Style Sheet zu laden. In diesem Style Sheet könnten dann also unterschiedliche Schriftgrößen für die jeweiligen Betriebssystem notiert werden. Das es sich bei diesem

Buch aber nicht um ein Style-Sheet-Buch handelt, soll auf diesen Aspekt nicht weiter eingegangen werden. Widmen wir uns vielmehr dem laufenden Betriebssystemtyp. Dieses lässt sich über die Eigenschaft **platform** ermitteln. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Betriebssystem()
{
document.Formular.Ausgabe.value = navigator.platform;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" value="Plattform" name="Ausgabe">
</FORM>
<A href="javascript:Betriebssystem()">Betriebssystem</A>
</BODY>
</HTML>
```

Listing 8.6: Das Betriebssystem wird im Eingabefeld angezeigt.

Innerhalb der gezeigten Syntax befindet sich ein Formular, in welches ein einzeiliges Eingabefeld integriert wurde. Unterhalb hiervon ist ein Hyperlink definiert, bei dessen Anklicken die Funktion `Betriebssystem()` aufgerufen wird. Diese schreibt den Typ des verwendeten Betriebssystems in das vorhandene Eingabefeld. Bei Windows-Systemen ist dies beispielsweise *Win32*.

8.1.7 HTTP-Identifikation

detaillierte Informationen

Innerhalb der HTTP1.1-Spezifikation wurde ein Schema festgelegt, in welchem die Anmeldung eines WWW-Browsers am Server beschrieben ist. Diese Vereinheitlichung macht vor allem im Hinblick auf Anwendungen, die sich der Unterscheidung der jeweiligen Browsertypen widmen, Sinn. So kann anhand der Informationen, die ein Browser an einen Webserver übermittelt, überprüft werden, um welchen Browser und um welche Version es sich handelt. Natürlich ist aber auch diese Vereinheitlichung bislang häufig nur in der Theorie umgesetzt wurden. Zwar hat sich mittlerweile ein relativ stimmiges Schema in der Praxis herauskristallisiert, bis hin zu einer vollständigen Übereinstimmung dürfte jedoch noch eine geraume Zeit vergehen. So ist es beispielsweise

innerhalb des Opera-Browsers möglich, die Browser-Art, mit der sich Opera am Server anmeldet, selbst zu definieren. Dass hieraus ungenaue Ergebnisse resultieren, muss an dieser Stelle nicht weiter ausgeführt werden. Nachfolgend finden Sie drei typische Meldungen dreier weit verbreiteter Browser. Sie werden auf den ersten Blick einige Gemeinsamkeiten, aber auch Unterschiede feststellen können.

Das erste Beispiel zeigt die Informationen des Internet Explorers 6.0 unter dem Betriebssystem Windows XP. Zunächst lässt sich nur wenig von dessen wahrer Identität erahnen. So gibt sich das Microsoft-Produkt zunächst als Mozilla aus. Durch das in Klammern folgende **compatible** wird diese Auszeichnung erörtert. Die 6er-Produktversion des Internet Explorers ist demnach mit dem Netscape Navigator 4.0 vollständig kompatibel. Dass dies nicht in Gänze stimmt, lässt sich leicht anhand des Einsatzes eines **<layer>**-Tags innerhalb des Internet Explorers überprüfen. Seine wahre Identität gibt der Internet Explorer erst innerhalb der Klammern bekannt. Hinter der Abkürzung MSIE folgt die Produktversion. Anschließend wird das Betriebssystem bekannt gegeben. Beachten Sie, dass die in Klammern angezeigten Informationen innerhalb der unterschiedlichen Browserversionen variieren.

```
Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;Q312461)
```

Opera als Sonderfall

Der Opera-Browser, so nett dieser Browser auch sonst daherkommen mag, genügt aus Sicht der gesendeten HTTP-Daten nur in einigen Fällen den Anforderungen. Auf den ersten Blick, und dies zeigt die folgende Opera-Meldung, erscheint die Meldung völlig korrekt. Opera meldet sich unter seinem richtigen Namen und mit der entsprechenden Versionsnummer. Zusätzlich wird die vom Browser verwendete Sprache ausgegeben. Da der Anwender innerhalb dieses Browsers jedoch selbst entscheiden kann, als welches Produkt sich Opera am Webserver anmelden soll, sind die gesendeten Daten häufig verfälscht. Die folgende Angaben gibt Opera bei korrekten Einstellungen unter dem Betriebssystem Windows XP bekannt:

```
Opera/6.01 (Windows XP; U) [en]
```

Der ehemalige Markführer, der Netscape Navigator, meldet sich am Web-Server mit dem Namen Mozilla an. Hinter dem Schrägstrich folgt die Produktversion. Innerhalb der Klammer stehen die Beschreibung des Betriebssystems sowie die verwendete Landessprache. Den tatsächlichen Namen, nämlich Netscape 6, gibt dieser Browser ganz am Ende der Meldung aus.

```
Mozilla/5.0 (Windows NT 5.1; de-DE;rv:0,9,4) Gecko/20011019 Netscape6/6.2
```

Auf Grund der unterschiedlichen Schemata, welche die Browser an den WWW-Server senden, ist eine korrekte Interpretation dieser Angaben gerade im Hinblick auf JavaScript-Anwendungen problematisch. Um die genannten Informationen vom Browser abzufragen, wird die Eigenschaft **userAgent** verwendet. Von einer Anwendung dieser Eigenschaft im Hinblick auf Browserweichen ist jedoch abzuraten, da die Ergebnisse nur mit viel Aufwand verwertbar aufbereitet werden können. Möchten Sie dennoch die **userAgent**-Eigenschaft für diesen Zweck nutzen, so müssen Sie den Inhalt der vom Browser gelieferten Zeichenkette durchsuchen und die betreffenden Elemente extrahieren. Das folgende Beispiel gibt alle Informationen, die der Browser bereit ist, bekannt zu geben, bei dem Aufruf der Seite innerhalb eines **alert()**-Meldungsfensters aus.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeDaten()
{
alert(navigator.userAgent);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeDaten()">Ihre Browserdaten</A>
</BODY>
</HTML>
```

Listing 8.7: Die HTTP-Identifikation wird in einem Meldungsfenster ausgegeben.

Die **userAgent**-Eigenschaft lässt sich zwar nicht so einfach für komplexe Abfragen nutzen, simple Anwendungen können jedoch leicht realisiert werden. So ist es beispielsweise möglich, einen bestimmten Browser zu erkennen und den Anwender auf die für ihn optimierten Seiten weiterzuleiten.

Abbildung 8.2: Der Internet Explorer 6 unter WindowsXP

8.1.8 Ist Java eingeschaltet?

Java ist weit verbreitet

Die steigende Popularität von Java in Verbindung mit der Kommunikation mit JavaScript machte die Einführung einer Methode notwendig, mit deren Hilfe überprüft werden kann, ob beim Anwender Java aktiviert ist. Diese Methode lautet **javaEnabled** und liefert als Ergebnis die boolschen Werte **true** und **false** zurück. Wobei **true** dann zurückgegeben wird, wenn Java verfügbar ist. Anderenfalls liefert der Browser **false**. Gerade vor dem Hintergrund, dass immer noch zahlreiche Anwender kein Java auf ihrem System zulassen, ist die **javaEnabled()**-Methode ein unverzichtbares Stilmittel, um möglichst viele Nutzer zufrieden stellen zu können. Eine diesbezügliche Anwendung beschreibt das folgende Beispiel:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
if(navigator.javaEnabled())
document.write("<APPLET code=\"../ps_light.class\" width=\"200\"
height=\"100\"><\/APPLET>");
else
document.write("<IMG src=\"../content.gif\" width=\"200\" height=\"100\">");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.8: Es wird entweder ein Java-Applet oder ein A-Gif angezeigt.

Anhand dieser Syntax bekommt der Anwender in jedem Fall ein für ihn optimiertes Ergebnis angezeigt. Innerhalb der **if**-Abfrage wird zunächst über **navigator.javaEnabled()** überprüft, ob der Anwender Java aktiviert hat. Wird hierbei der Wahrheitswert **true** zurückgeliefert, wird in das Dokument dynamisch über das **<APPLET>**-Tag ein Java-Applet eingebunden. Für den Fall, dass diese **if**-Bedingung nicht erfüllt, der Rückgabewert also **false** ist, wird statt des Java-Applets eine Grafik eingebunden. So kommt zwar der Anwender nicht in den vollen Genuss Ihrer Programmierkenntnisse, erhält aber dennoch mehr als nur ein leeres Browserfenster.

8.2 Alles über den Bildschirm

Elemente in Abhängigkeit von der Bildschirmgröße anzeigen

Durch das **screen**-Objekt können Sie Informationen über den vom Anwender verwendeten Monitor einholen. Hierdurch wird es möglich, Ihre Seiten für verschiedene Monitore zu optimieren. So können Sie beispielsweise in Abhängigkeit von der Bildschirmgröße festlegen, ob innerhalb eines Framesets Bildlaufleisten angezeigt werden sollen. Sie können folgendermaßen auf die Eigenschaften des **screen**-Objekts zugreifen:

```
screen.Eigenschaft
```

Das hier keine Syntax für Methoden aufgeführt sind, liegt daran, dass das **screen**-Objekt keine Methoden kennt. Auf die Eigenschaft **height**, mit der die Höhe des Monitors ermittelt wird, können Sie folgendermaßen zugreifen:

```
screen.height
```

Optimierung von Internetprojekten

Dass **screen**-Objekt eignet sich vorzüglich, um Internetseiten für bestimmte Bildschirme zu optimieren. Beachten Sie jedoch, dass hierfür optimierte Seiten häufig ein Mehr an Aufwand bedeuten. Überprüfen Sie beispielsweise die Monitorgröße und lassen in deren Abhängigkeit jeweils eine andere Seite laden, müssten Sie mehrere Versionen einer Seite erstellen. Es ist demnach zu überdenken, ob sich dieser Aufwand tatsächlich lohnt.

8.2.1 Maximale Anzeigehöhe

die tatsächlich nutzbare Anzeigehöhe

Die **availHeight**-Eigenschaft ermöglicht das Auslesen der maximal zur Verfügung stehenden Bildschirmhöhe. Hierbei ist es unerheblich, ob die aktuelle Browserinstanz maximiert dargestellt wird oder nicht. Vielmehr interpretiert der Browser die Höhe, die eine Anwendung im Vollbildmodus einnehmen könnte. Als Einschränkung müssen hierbei alle konstant angezeigten Bildschirmelemente genannt werden. So hat beispielsweise das permanente Vorhandensein der Taskleiste zur Folge, dass der von dieser Leiste beanspruchte Platz von der **availHeight**-Eigenschaft von der Gesamthöhe abgezogen wird. Um die tatsächlich zur Verfügung stehende Gesamthöhe auslesen zu können, eignet sich die **availHeight**-Eigenschaft demnach nur bedingt. Besser geeignet für diesbezüglich konkretere Angaben ist der Einsatz der **screen**-Eigenschaft. Weiterführende Informationen hierzu finden Sie im vorigen Abschnitt. Nichtsdestotrotz lässt sich aber auch die **availHeight**-Eigenschaft für praxistaugliche Anwendungen nutzen. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function FarbenHoehe()
{
var Farbe = window.screen.colorDepth;
var Hoehe = window.screen.availWidth;
if(Farbe == 16 || Farbe == 32)
{
alert("maximale Höhe: " + Hoehe + "\n Farbtiefe: " + Farbe + " Bit");
}
else if(Farbe == 8)
{
alert("maximale Höhe: " + Hoehe + "\n Sehr wenig Farben!");
}
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:FarbenHoehe()">Höhe und Farbtiefe
ermitteln</A>
</BODY>
</HTML>
```

Listing 8.9: Die Farbtiefe und Anzeigehöhe werden in einem Meldungsfenster ausgegeben.

Die Grundlage dieses Scripts ist die Abfrage der bei dem Anwender auslesbaren **colorDepth**-Eigenschaft, auf die im Laufe dieses Abschnitts aber noch genauer eingegangen wird. Weitere Informationen zu **colorDepth** finden Sie im Abschnitt 8.2.3. Nach der jeweiligen Abfrage der **colorDepth**-Werte wird in einem Meldungsfenster die bei dem Anwender zur Verfügung stehende Höhe des Anzeigebereichs in einem Meldungsfenster ausgegeben. Praktische Anwendungen lassen sich mit **availHeight** vor allem immer dann realisieren, wenn Internetseiten für bestimmte Auflösungen optimiert sind. So könnte beispielsweise die zur Verfügung stehende Gesamthöhe ausgelesen, und anschließend eine hierfür geeignete Seite geladen werden.

8.2.2 Maximale Anzeigebreite

die tatsächlich nutzbare Anzeigehöhe

Anhand der **availWidth**-Eigenschaft kann die beim Anwender maximal zur Verfügung stehende Bildschirmbreite gespeichert werden. Gemessen wird die Breite, die eine entsprechende Anwendung im Vollbildmodus ausfüllen kann. Hierbei werden alle feststehenden Bildschirmele-

mente, wie beispielsweise die Taskleiste, von der eigentlichen Bildschirmbreite abgezogen. Das bedeutet, dass bei einer eigentlichen Bildschirmbreite von 1024 Pixel nicht dieser Wert, sondern bei einer seitlich angeordneten und sichtbaren Taskleiste ein Wert von nur etwa 970 Pixel durch **availWidth** gespeichert wird. Eingesetzt werden kann die **availWidth**-Eigenschaft beispielsweise für die Realisierung eines Vollbildmodus von Internetseiten. Und einen ebensolchen Fall beschreibt die nachfolgende Syntax:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Vollbild(URL,Fenstername) {
        if (window.screen) {
        var Breite = screen.availWidth;
        var Hoehe= screen.availHeight;
 go = window.open(URL,Fenstername,'width=' + Breite
+ ',height=' +
Hoehe + ',fullscreen=1,scrollbars=1,left=' + (0) +
',top=' + (0));
}
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Vollbild('http://www.offspring.com,
Fenster');">The Offspring im Vollbild-Modus</A>
</BODY>
</HTML>
```

Listing 8.10: Die aufgerufene Datei wird im Vollbild-Modus angezeigt.

Innerhalb des Dateikörpers wurde ein Verweis definiert, bei dessen Anklicken die Funktion `Vollbild()` aufgerufen wird. In dieser Funktion werden die beiden Variablen `Breite` und `Hoehe` deklariert. Wobei für diesen Abschnitt besonders die Variable `Breite` von Bedeutung ist. Bedient sich diese doch der **avialWidth**-Eigenschaft. Im nächsten Schritt wird das globale Erscheinungsbild des neu zu öffnenden Fensters spezifiziert. Neben den bereits bekannten Eigenschaften wie beispielsweise **scrollbars** werden hier auch die Breite und Höhe des Fensters festgelegt. Dieses nimmt automatisch den maximal zur Verfügung stehenden Anzeigebereich ein. Beachten Sie, dass die Anweisung **fullscreen=1** ebenfalls die Darstellung eines Vollbildmodus zur Folge hat. Dies gilt jedoch lediglich für den Internet Explorer, während die gesamte Syntax sich in dieser Form

parallel hierzu auch für den Netscape Navigator einsetzen lässt. Einziges Manko hinsichtlich des Netscape Navigators ist das Vorhandensein der Titelleiste innerhalb des neuen Fensters, die sich auch nicht für den Einsatz des gezeigten Beispiels ausblenden lässt.

8.2.3 Bildschirmhöhe

z.B. 600, 768 und 864 Pixel

Die **height**-Eigenschaft speichert die Höhe der Bildschirmauflösung. Es wird hierbei ein numerischer Wert wie beispielsweise 600, 768 oder 864 zurückgeliefert. Der ermittelte Wert kann für die Darstellung von Internetseiten von entscheidender Bedeutung sein. Dies gilt vor allem für solche Seiten, die für eine bestimmte Bildschirmgröße optimiert dargestellt werden sollen. So lässt sich durch die Abfrage der Höhe beispielsweise ermitteln, in welcher Größe ein neues Fenster geöffnet werden soll. Gleichfalls ist es aber auch denkbar, dass je nach Höhe der Bildschirmauflösung eine automatische Weiterleitung auf eine hierfür optimierte Seite erfolgt. Wie sich dies realisieren lässt, beschreibt die nachfolgende Syntax.

```
<HTML>
<HEAD>
</HEAD>
<SCRIPT type="text/JavaScript">
   if(screen.height >= 600)
      window.location.href = "gross.htm"
   else
      window.location.href = "klein.htm";
</SCRIPT>
<BODY>
</BODY>
<HTML>
```

Listing 8.11: Je nach Bildschirmauflösung wird eine hierfür optimierte Seite angezeigt.

Innerhalb des direkt mit dem Einlesen der Datei ausgeführten Script-Bereichs wird mittels einer **if**-Abfrage die Höhe der Bildschirmauflösung ermittelt. Ist diese größer oder gleich `600`, erfolgt eine automatische Weiterleitung auf die Datei `gross.htm`. Anderenfalls, also wenn die Höhe kleiner als `600` ist, wird automatisch auf die Datei `klein.htm` weitergeleitet. Selbstverständlich ließe sich dieses Beispiel noch ausbauen. So könnten alle gängigen Höhen abgefragt und eine jeweils andere Seite geladen werden. Ob dieses Vorgehen sinnvoll wäre, mag an dieser Stelle freilich bezweifelt werden.

8.2.4 Bildschirmbreite

Internetseiten standardisieren

Die Eigenschaft **width** ermöglicht das Speichern der absolut zur Verfügung stehenden Breite des Bildschirms in Pixel. Da diese Breite im Normalfall standardisiert ist, lässt sich diese vortrefflich für den Einsatz innerhalb von auf JavaScript basierenden Abfragen nutzen. Häufig vorkommende Werte im Zusammenhang mit der **width**-Eigenschaft sind 640, 800 und 1024. Beachten Sie, dass die **width**-Eigenschaft die Gesamtbreite des Monitors ausliest und sich nicht etwa auf die Breite der aktuellen Browserinstanz beruft. Darüber hinaus werden von dieser Eigenschaft keine feststehenden Bereiche, wie etwa die Taskleiste, berücksichtigt. Vielmehr werden auch diese in die Berechnung der Gesamtbreite mit einbezogen und nicht wie im Zusammenhang mit der **availWidth**-Eigenschaft als separate Elemente betrachtet. Dass die **width**-Eigenschaft nicht nur im Umgang mit dem Laden von für bestimmte Bildschirmauflösungen optimierten Seiten geeignet ist, soll exemplarisch die folgende Syntax veranschaulichen:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
if (navigator.appVersion.substring(0,1) >= "4")
{
  if ( screen.width > "1024" ) {
    document.write("<IMG src=\"gross.gif\" width=\"300\" height=\"300\">");
  } else if ( screen.width == 800 ) {
    document.write("<IMG src=\"mittel.gif\" width=\"200\" height=\"200\">");
  } else {
    document.write("<IMG src=\"klein.gif\" width=\"100\" height=\"100\">");
  }
}
//-->
</SCRIPT>
</BODY>
<HTML>
```

Listing 8.12: Je nach Bildschirmbreite wird eine hierfür optimierte Grafik angezeigt.

Das aufgeführte Beispiel beschreibt die Integration einer Grafik über ein JavaScript. Bis hierin erscheint dies freilich noch nicht sonderlich erbauend zu sein. Interessant wird das gezeigte Script allerdings dann, wenn betrachtet wird, dass nicht immer die gleiche, sondern die für jeweils eine andere Bildschirmbreite optimierte Grafik geladen wird. So wird beispielsweise bei einer Gesamtbreite des Anwenderbildschirms

von 1024 Pixel die Grafik gross.gif geladen. Es wird angenommen, dass diese Grafik eben exakt für diese Bildschirmbreite konzipiert wurde. Insgesamt werden in dem Beispiel drei mögliche Gesamtbreiten, nämlich 1024 Pixel, 800 Pixel, oder ein anderer als diese beiden Werte, abgefragt. Abhängig von der ausgelesenen Bildschirmbreite wird die hierfür optimierte Grafik angezeigt.

8.2.5 Farbtiefe

Bit-Anzahl

Die Eigenschaft **colorDepth** erlaubt es herauszufinden, welche Bit-Anzahl auf dem Bildschirm des Nutzers für die Darstellung einer Farbe an jedem Pixelpunkt verwendet wird. Beachten Sie, dass nur dann ein sinnvoller Wert ausgelesen werden kann, wenn der Bildschirm intern eine Farbpalette verwendet. Anderenfalls wird im Internet Explorer der Wert **null** und beim Netscape Navigator der Wert **undefined** zurückgegeben. Im nachstehenden Beispiel wird ein neues Fenster geöffnet und hierin die entsprechende Bit-Anzahl angezeigt.

```
<HTML>
<HEAD>
</HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Farbtiefe()
{
 var Fenster;
 var Tiefe;
Fenster=window.open("","Tiefe","width=300,height=200,resizable=no");
Tiefe=screen.colorDepth;
Fenster.document.write("<H3>Die aktuelle Bit-Anzahl beträgt:
" + Tiefe + "</H3>");
}
//-->
</SCRIPT>
<BODY onload="Farbtiefe()">
</BODY>
<HTML>
```

Listing 8.13: Die Bit-Anzahl wird dynamisch in das neue Fenster geschrieben.

Ausgelöst wird die Funktion Farbtiefe() durch den Event-Handler **onload**. Innerhalb der Funktion wird über die **open()**-Methode ein neues Fenster geöffnet. Anschließend wird der Variablen Tiefe die Anweisung **screen.colorDepth** zugewiesen. Der Wert dieser Variablen wird über die **write()**-Methode dynamisch in das Dokument geschrieben.

Abbildung 8.3: Die Bit-Anzahl beträgt hier 32.

8.2.6 Farbauflösung

Übliche Werte sind 16 und 32 Bit

Die Eigenschaft **pixelDepth** speichert die verfügbare Farbtiefe des Anwenderbildschirms. Ausgegeben wird ein Wert, welcher sich auf die Bits pro Pixel bezieht. Gängige Werte sind hierbei beispielsweise *16* oder *32*. Beachten Sie, dass nur dann ein gültiger Wert zurückgeliefert wird, wenn der Anwenderbildschirm eine interne Farbpalette benutzt. Anderenfalls wird innerhalb des Netscape Navigators kein Wert, sondern **undifened** ausgegeben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
if(screen.pixelDepth)
 document.write(screen.pixelDepth + " Bit");
else
 alert("Leider nicht verfügbar");
//-->
</SCRIPT>
</BODY>
<HTML>
```

Listing 8.14: Die Farbauflösung wird in das Dokument geschrieben.

In dem gezeigten Beispiel wird zunächst überprüft, ob der Eigenschaft **pixelDepth** ein Wert zugewiesen werden kann. Ist dies der Fall, wird dieser dynamisch in das Dokument geschrieben. Sollte kein Wert verfügbar sein, öffnet sich ein Meldungsfenster mit einem entsprechenden Hinweistext.

8.2.7 Hintergrundgrafik nach Bildschirmbreite

Kachelung verhindern

Hintergrundgrafiken stellen häufig ein Problem dar. So kommt es beispielsweise oft vor, dass je nach Größe des Anzeigefensters die Grafik gekachelt, also so oft wiederholt wird, bis der gesamte Anzeigebereich ausgefüllt ist. Bei dezenten Hintergrundgrafiken mag dies noch hinnehmbar sein, für auffällige Grafiken ist dieses Prinzip jedoch nicht geeignet. Zwar kann beispielsweise im Internet Explorer die Kachelung durch das **fixed**-Attribut im **<BODY>**-Tag verhindert werden, eine optimale Lösung ist aber auch das nicht. Abhilfe kann das folgende JavaScript schaffen. Hier wird die Breite des Anwenderbildschirms abgefragt. Je nachdem, welcher Wert hier gespeichert wurde, wird eine entsprechend breite Hintergrundgrafik geladen.

```
<HTML>
<HEAD>
</HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Fensterbreite = screen.width;
if (Fensterbreite >= 1280)
document.write('<BODY background="image1.gif">');
else
if (Fensterbreite >= 1024)
document.write('<BODY background="image2.gif">');
else
if (Fensterbreite <= 800)
document.write('<BODY background="image3.gif">');
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.15: In Abhängigkeit von der Bildschirmbreite wird eine entsprechende Hintergrundgrafik angezeigt.

In der Variablen `Fensterbreite` wird über die **width**-Eigenschaft des **screen**-Objekts die Bildschirmbreite gespeichert. Anschließend wird mittels bedingter **if**-Abfragen überprüft, ob dieser Wert größer oder gleich `1280`, `1024` oder `800` ist. Je nach Rückgabewert wird über die **write()**-Methode die entsprechende Hintergrundgrafik eingebunden. Hierfür werden das **<BODY>**-Tag, dessen **background**-Attribut sowie der Name der Grafik notiert.

8.3 Alles über Plug-Ins

im Internet Explorer nicht verfügbar

Beachten Sie, dass der Einsatz des **plugin**-Objekts nur im Zusammenhang mit dem Netscape Navigator sinnvoll ist. Durch Plug-Ins wird dem Programmierer eine Schnittstelle zur Verfügung gestellt, durch die der Funktionsumfang des Navigators erweitert werden kann. Somit lassen sich beispielsweise ganz bestimmte Grafikformate zur Wiedergabe von Animationen verwenden. So vorteilhaft dies auch sein mag, diese Technologie hat dennoch enorme Nachteile. Können Dateien nur mittels eines Plug-Ins wiedergegeben werden, streiken hier andere Browser. Die in eine Seite integrierten Dateien können somit häufig nicht abgespielt bzw. angezeigt werden. Ein weiterer Nachteil ergibt sich aus der Akzeptanz von Plug-Ins beim Anwender. Nur die wenigsten Nutzer des Netscape Navigators werden sich für jede kleine Animation, die ein eigenes Plug-In benötigt, dieses herunterladen. Entscheiden Sie also vor dem Einsatz eines exotischen Dateiformats, welches ein Plug-In benötigt, ob Sie sich nicht besser einer etablierten Lösung bedienen. Auf das **plugins**-Objekt und dessen Eigenschaften kann folgendermaßen zugegriffen werden:

```
navigator.plugins[#].Eigenschaft
navigator.plugins["Plugin-Name"].Eigenschaft
```

Sie können auf das **plugins**-Objekt über eine Indexnummer oder dessen Namen zugreifen. Bei der Verwendung einer Indexnummer muss zunächst **navigator.plugins** und anschließend in eckigen Klammern eine Ziffer notiert werden. Bei der Verwendung der Ziffer `3` würden Sie also beispielsweise das vierte Plug-In ansprechen. Da Sie jedoch nicht wissen können, an welcher Stelle das gesuchte Plug-In gespeichert ist, sollte auf die Verwendung von Indexnummern verzichtet werden. Im Umgang mit Plug-Ins ist die Verwendung des Plug-In-Namens die bessere Variante. Auch hierbei wird zunächst die Anweisung **navigator.plugins** notiert. In eckigen Klammern wird der Name des Plug-Ins angegeben. Zwei Beispiele:

```
navigator.plugins[2].filename
navigator.plugins["LiveAudio "].filename
```

Durch die erste Zeile wird auf das dritte gespeicherte Plug-In und dessen Eigenschaft **filename** zugegriffen. Wie erwähnt, wissen Sie zunächst nicht, welches Plug-In sich an der dritten Position befindet. Besser ist hier also der Einsatz der zweiten Syntax. Diese greift über den Plug-In-Namen `LiveAudio` auf das `LiveAudio`-Plug-In und dessen **filename**-Eigenschaft zu. Beachten Sie, dass das **plugins**-Objekt nur Eigenschaften, aber keine Methoden kennt.

8.3.1 Kurzbeschreibung

Art und Verwendungszweck bestimmen

Mittels der **description**-Eigenschaft kann eine Kurzbeschreibung vorhandener Plug-Ins gespeichert werden. Der hierbei zurückgelieferte Wert hilft dabei, die Art und den Verwendungszweck des Plug-Ins rasch erkennen zu können. Wird beispielsweise die Kurzbeschreibung des QuickTime-Plug-Ins angefordert, wird folgender Beschreibungstext ausgegeben. "The QuickTime Plug-In allows you to view a wide variety of multimedia content in Web pages. For more information, visit the QuickTime Web site."

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
if(navigator.plugins["LiveAudio"])
document.write(navigator.plugins['LiveAudio'].description);
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 8.16: Beschreibung des Plug-Ins

Im aufgeführten Beispiel wird die Kurzbeschreibung des `LiveAudio`-Plug-Ins dynamisch in das Dokument geschrieben. Hierzu wird zunächst überprüft, ob das Plug-In auf dem Rechner des Anwenders installiert ist. Ist dies der Fall, wird über **navigator.plugins[,LiveAudio']** auf das Plug-In zugegriffen und dessen Kurbeschreibung mittels der **description**-Eigenschaft ermittelt. Der zurückgelieferte Wert wird über die **write()**-Methode ausgegeben.

8.3.2 Dateiname

Name der Programmdatei

Der Dateiname eines Plug-Ins kann über die **filename**-Eigenschaft ermittelt werden. Beachten Sie, dass hierbei als Rückgabewert jeweils der Name der Programmdatei einschließlich absoluter Pfadangabe geliefert wird. Ein typischer Wert ist hier beispielsweise C:\Programme\Netscape\Communicator\Program\plugins\npwmsdrm.dll. Anhand der nachstehenden Syntax werden alle auf dem System des Anwenders installierten Plug-Ins ermittelt und deren jeweilige Programmdatei wird einschließlich der Pfadangabe dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
</HEAD>
```

```
<BODY>
<SCRIPT type="text/JavaScript">
for(i=0; i<navigator.plugins.length; ++ i)
document.write("<BR><i>" + navigator.plugins[i].filename);
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.17: Ausgabe aller Dateinamen

Anhand einer **for**-Schleife werden alle vorhanden Plug-Ins ausgelesen. Die ermittelten Werte werden mittels der **write()**-Methode ausgegeben. Zur optischen Auflockerung wird jedes Plug-In in kursiver Schrift sowie jeweils in einer neuen Zeile angezeigt.

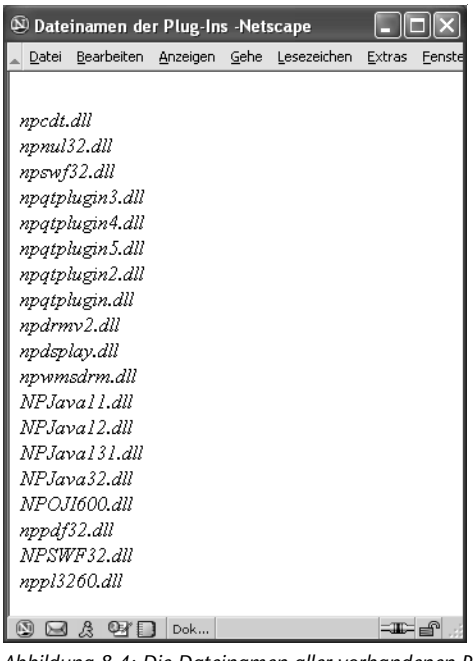


Abbildung 8.4: Die Dateinamen aller vorhandenen Plug-Ins

8.3.3 Anzahl

Die Anzahl der installierten Plug-Ins lässt sich durch den Einsatz der **length**-Eigenschaft ermitteln. Als Rückgabe wird ein numerischer Wert geliefert. Anhand der nachstehenden Syntax wird die Anzahl der vorhandenen Plug-Ins ermittelt und dieser Wert in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
alert(navigator.plugins.length);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.18: Ausgabe in einem Meldungsfenster

Der integrierte Script-Bereich wird direkt mit dem Einlesen der Datei ausgeführt. Hierin wird über **navigator.plugins** auf die Plug-Ins zugegriffen. Wie viele Plug-Ins letztendlich installiert sind, wird über die **length**-Eigenschaft ermittelt. Die Ausgabe erfolgt mittels der **alert()**-Methode.

8.3.4 Name

Anhand der **name**-Eigenschaft kann der Name eines installierten Plug-Ins gespeichert werden. Die Verwendung dieser Eigenschaft ist vor allem immer dann sinnvoll, wenn überprüft werden soll, ob ein benötigtes Plug-In bereits installiert ist. Ein Beispiel:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<H3>Folgende Plug-Ins sind installiert:</H3>
<SCRIPT type="text/JavaScript">
for(i=0; i<navigator.plugins.length; ++ i)
document.write("<BR><i>" + navigator.plugins[i].name);
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.19: Die Namen werden dynamisch in das Dokument geschrieben.

Der Dateikörper enthält einen Script-Bereich. In diesem werden anhand einer **for**-Schleife alle auf dem System installierten Plug-Ins ermittelt und deren Namen werden jeweils innerhalb einer neuen Zeile in kursiver Schrift ausgegeben.

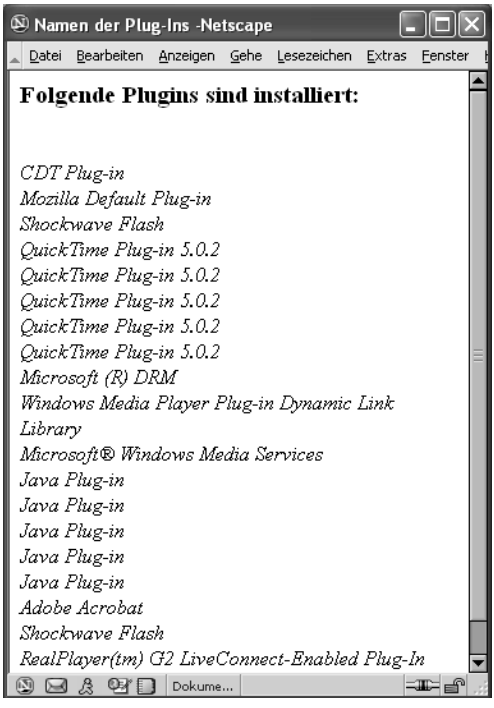


Abbildung 8.5: Die Namen der installierten Plug-Ins werden ausgegeben.

8.4 Alles über MIME-Typen

MIME-Typen bestehen aus einem Medien- und einem Subtyp

Jeder Webbrowser und Server kennt eine bestimmte Anzahl an MIME-Typen. Der grundsätzliche Aufbau von MIME-Typen erfolgt stets in zwei Teilen: Zuerst wird der Medien- und, durch einen Schrägstrich getrennt, dann der Subtyp angegeben. Der MIME-Typ für HTML-Dateien wird beispielsweise mit **`text/html`** angegeben. Über das **`mimeTypes`**-Objekt können Sie überprüfen, welche Datentypen vom Browser des Anwenders erkannt werden. Ebenso lässt sich hierdurch feststellen, ob ein bestimmtes Plug-In installiert ist. Sie können das **`mimeTypes`**-Objekt durch zwei unterschiedliche Syntaxformen verwenden. Zur Auswahl stehen hier der Zugriff durch eine Indexnummer und durch einen Indexnamen.

```
navigator.mimeTypes[#].Eigenschaft
navigator.mimeTypes["Mime-Type"].Eigenschaft
```

Um auf einen MIME-Typ über eine Indexnummer zuzugreifen, wird die Anweisung **`navigator.mimeTypes`** notiert. In den eckigen Klammern wird eine Zahl notiert. Über diese Zahl sprechen Sie den MIME-Typ an. Hieran schließt sich die Notation der gewünschten Eigenschaft an. Methoden sind für das **`mimeTypes`**-Objekt nicht verfügbar. Da Sie nicht wis-

sen, an welcher Position ein MIME-Typ verfügbar ist, eignet sich diese Syntaxform nur bedingt. Effizienter ist der Zugriff über einen Indexnamen. Auch hierbei wird zunächst die Anweisung **navigator.mimeTypes** notiert. Innerhalb der eckigen Klammern wird der Name des MIME-Typs angegeben. Für eine HTML-Datei lautet dieser **text/html**. Im Anschluss hieran wird die gewünschte Eigenschaft des **mimeTypes-Objekts** angegeben. Die beiden folgenden Beispiele sollen das zuvor Beschriebene veranschaulichen.

```
navigator.mimeTypes[3].description
navigator.mimeTypes["application/pdf"].description
```

Durch die erste Zeile wird auf den vierten vorhandenen MIME-Typ zugegriffen. Interessanter ist die zweite Notationsform. Hierbei wird als MIME-Typ **application/pdf** angegeben. Dabei handelt es sich um den MIME-Typ des weit verbreiteten PDF-Formats. Durch die **description**-Eigenschaft lässt sich eine Beschreibung dieses MIME-Typs ausgeben.

8.4.1 Beschreibung

Die Kurzbeschreibung eines MIME-Typs lässt sich über die Eigenschaft **description()** auslesen. Im folgenden Beispiel wird die Kurzbeschreibung von `image/gif` in einem Meldungsfenster ausgegeben:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
alert(navigator.mimeTypes["image/gif"].description);
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 8.20: Die Kurzbeschreibung wird in einem Meldungsfenster ausgegeben.

Der Script-Bereich wird mit dem Einlesen der Datei ausgeführt. Der auszulesende MIME-Typ lautet hier `image/gif`. Die Kurzbeschreibung wird mittels der **alert()**-Methode ausgegeben. Die zurückgelieferte Kurzbeschreibung ist *GIF Image*.

8.4.2 Ist ein bestimmtes Plug-In vorhanden?

Flash als Beispiel

Ob der Browser einen bestimmten MIME-Typ kennt, lässt sich über die **enabledPlugin**-Eigenschaft ermitteln. Sinnvoll ist der Einsatz dieser Eigenschaft wenn man sich vor Augen führt, dass in der heutigen Zeit

viele Projekte Flash-Elemente benutzen. So werden beispielsweise Werbebanner häufig als Flash-Objekte eingebunden. Anwender die keinen oder nicht den neuesten Flash-Player besitzen, bekommen hier eine Fehlermeldung angezeigt. Sinnvoller ist es in einem solchen Fall, zwei Versionen der Werbebotschaft, nämlich einmal in Flash und einmal als Animated GIF, anzubieten. Da der Anwender hier in der Regel nicht selbst entscheiden sollte, welche Version ihm angeboten wird, kann über die **enabledPlugin**-Eigenschaft überprüft werden, ob das Flash-Plug-In installiert ist. Ist dies der Fall, wird die Werbung Flash-basiert, anderenfalls als Animated GIF angezeigt. Mit dem beschriebenen Fall befasst sich das folgende Beispiel:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
if(navigator.mimeTypes["application/x-shockwave-flash"].
enabledPlugin != null)
{
document.write('<object classid="CLSID:D27CDB6E-AE6D-11cf-96B8444553540000"')
document.write('codebase="http://active.macromedia.com/flash2/cabs/
swflash.cab#version=4,0,0,0" width="300" height="200">')
document.write('<param name="movie" VALUE="werbung.swf"><param name="bgcolor"
value="#ffffff">')
document.write('</object>');
}
else
document.write('<img src="werbung.gif">')
 //-->
 </SCRIPT>
</BODY>
</HTML>
```

Listing 8.21: Flash oder Animated GIF?

In der aufgeführten Syntax wird in einer **if**-Abfrage überprüft, ob das Shockwave-Plug-In vorhanden ist. Ist dies der Fall, wird über mehrere **write()**-Anweisungen die Flash-Datei `werbung.swf` eingebunden. Hierzu wird das **<object>**-Tag mit diversen Parametern verwendet. Der **else**-Zweig beinhaltet den Code, der für den Fall ausgeführt wird, dass das Shockwave-Plug-In nicht installiert ist. In diesem Fall wird statt der Datei `werbung.swf` die Animated GIF-Datei `werbung.gif` angezeigt. Auch deren Integration erfolgt mittels der **write()**-Methode.

8.4.3 Anzahl der MIME-Typen

Die Anzahl der MIME-Typen, welche vom Browser verarbeitet werden können, wird über die **length**-Eigenschaft ermittelt. Als Rückgabe wird ein numerischer Wert geliefert. Im folgenden Beispiel werden die dem Browser bekannten Dateiendungen untereinander dynamisch in das Dokument geschrieben. Um die Programmierung einer Endlosschleife zu verhindern, wird die **length**-Eigenschaft verwendet.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<H3>Folgende MIME-Typen sind auf Ihrem System verfügbar:</H3>
<SCRIPT type="text/JavaScript">
for(i=0; i<navigator.mimeTypes.length; ++i)
document.write("<BR>" + navigator.mimeTypes[i].suffixes);
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.22: Alle verfügbaren MIME-Typen werden untereinander in das Dokument geschrieben.

Anhand einer **for**-Schleife werden alle dem Browser bekannten Dateiendungen ausgelesen. Hierzu wird die **suffixes**-Eigenschaft verwendet. Um eine Endlosschleife zu verhindern, werden nur so viele Schleifendurchläufe ausgeführt, wie es die Anzahl der bekannten MIME-Typen zulässt.

8.4.4 Dateiendungen

mehrere Dateiendungen

Über die **suffixes**-Eigenschaft kann die Dateiendung bzw. können die Dateiendungen eines MIME-Typs ermittelt werden. Viele MIME-Typen können unterschiedliche Dateiendungen besitzen. So besitzt das Grafikformat JPEG beispielsweise die beiden möglichen Dateiendungen jpg und jpeg. Anhand der nachstehenden Syntax werden die beiden Dateiendungen des Grafikformats TIFF dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
document.write(navigator.mimeTypes["image/tiff"].suffixes);
</SCRIPT>
</HEAD>
<BODY>
```

```
</BODY>
</HTML>
```

Listing 8.23: Die Dateiendungen des MIME-Typs image/tiff werden in das Dokument geschrieben.

Der Script-Bereich wird mit dem Einlesen der Datei ausgeführt. Der übergebene MIME-Typ ist hier `image/tiff`. Die Werte, die in das Dokument geschrieben werden, sind *`tiff`* und *`tif`*.

8.4.5 MIME-Typ

Kategorie plus Typ

Die **`type`**-Eigenschaft ermittelt den Typ eines MIME-Typs. Zusätzlich zu dem Typ wird dessen Kategorie gespeichert. So würde beispielsweise bei dem Zugriff auf den MIME-Typ `pdf` der Rückgabewert `application/pdf` gespeichert werden, wobei die Kategorie hier **`application`** und der MIME-Typ `pdf` ist. Getrennt werden Kategorie und MIME-Typ durch einen Schrägstrich. Die nachstehende Syntax dient dazu, alle bekannten MIME-Typen zu ermitteln und diese dynamisch untereinander in das Dokument zu schreiben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<B>Folgende MIME-Typen kann Ihr Browser verstehen</B>
<SCRIPT type="text/JavaScript">
for(i=0; i<navigator.mimeTypes.length; ++i)
document.write("<BR><i>" + navigator.mimeTypes[i].type);
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.24: Alle MIME-Typen werden untereinander ausgegeben.

Welche MIME-Typen bekannt sind, wird anhand einer **`for`**-Schleife ermittelt. Um die Programmierung einer Endlosschleife zu verhindern, wird die **`length`**-Eigenschaft verwendet. Dies sorgt dafür, dass die Schleife nur so oft durchlaufen wird, wie es auch tatsächlich notwendig ist.

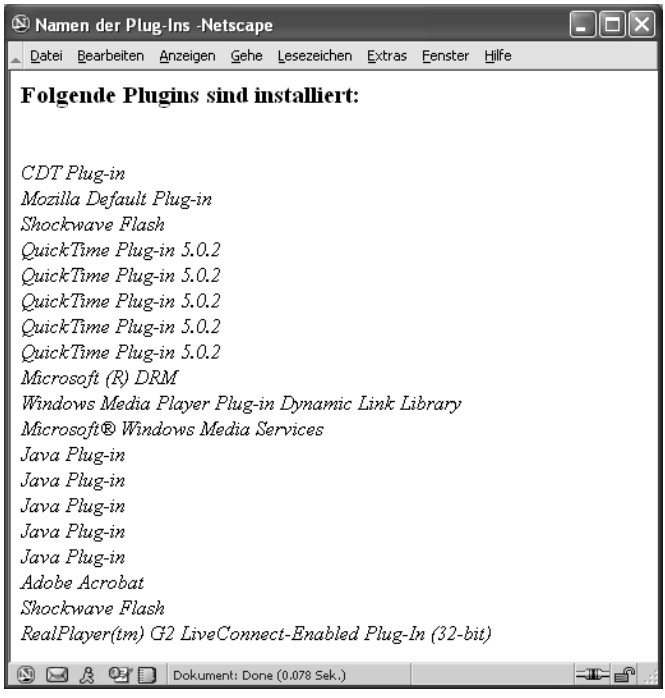


Abbildung 8.6: Alle dem Navigator 4 bekannten MIME-Typen

8.4.6 Alle MIME-Typen auslesen

mehrere Angaben kombinieren

Im folgenden Beispiel werden einige Eigenschaften des **mimeTypes**-Objekts miteinander kombiniert. Als Ergebnis erhalten Sie eine Tabelle, in der alle verfügbaren Plug-Ins aufgelistet werden. Für Sie als JavaScript-Programmierer ist dies vor allem bezüglich des Testens der bereits beschriebenen Eigenschaften interessant. Können Sie sich doch alle bei Ihnen installierten Plug-Ins anzeigen lassen und somit die Eigenschaften des **mimeTypes**-Objekts testen. In der erzeugten Tabelle werden der MIME-Typ-Bezeichner, das Dateisuffix und eine Beschreibung aufgelistet. In der letzten Spalte wird angezeigt, ob ein Plug-In installiert ist, mit dem die Dateien des entsprechenden Typs verarbeitet werden können.

```
<HTML>
<HEAD>
</HEAD>
<SCRIPT type="text/JavaScript">
<!--
document.writeln("<TABLE>");
 for( i = 0 ; i < navigator.mimeTypes.length;i++)
{ document.writeln("<TR>");
document.writeln("<TD>" + navigator.mimeTypes[i].type + "<\/TD>");
```

```
document.writeln("<TD>" + navigator.mimeTypes[i].suffixes  + "<\/TD>");
document.writeln("<TD>" + navigator.mimeTypes[i].description + "<\/TD>");
document.writeln("<TD> " + navigator.mimeTypes[i].enabledPlugin + "<\/
TD>"); document.writeln("<\/TR>"); }
document.writeln("<\/TABLE>");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 8.25: Die Ausgabe aller MIME-Typen in Tabellenform

Die notwendigen Tags für die Erstellung der Tabelle werden über die **writeln()**-Methode in das Dokument geschrieben. Mittels einer **for**-Schleife werden alle MIME-Typen, die im Browser verfügbar sind, angesprochen. Um die Programmierung einer Endlosschleife zu verhindern, benötigen wir ein Abbruchkriterium. Hierfür bedienen wir uns der **length**-Eigenschaft, durch welche die Anzahl der vorhandenen MIME-Typen als numerischer Wert gespeichert wird. Die **for**-Schleife wird demnach so lange durchlaufen, bis alle MIME-Typen angesprochen wurden. In den jeweiligen Tabellenspalten werden die entsprechenden Eigenschaften des **mimeTypes**-Objekts notiert.

8.5 Fragen und Übungen

1. Schreiben Sie eine Funktion, mit der Sie einen Browser der 4. und der 5. Generation erkennen können. Die Browsergeneration soll in einem Meldungsfenster ausgegeben werden. Tipp: Um die Browsergeneration zu ermitteln, müssen Sie mittels der **substring()**-Methode eine Teilzeichenkette aus dem zurückgelieferten Wert extrahieren. Die **substring()**-Methode erwartet zwei Parameter. Der erste gibt die Position des ersten zu extrahierenden Zeichens und der zweite die Position des ersten nicht mehr zu extrahierenden Zeichens an. Finden Sie eine geeignete Methode, um die Produktversion des Browsers zu erfahren. Erst anschließend sollten Sie mit der Extrahierung beginnen.

2. Warum ist die **appCodeName**-Eigenschaft des **navigator**-Objekts kein geeignetes Mittel, um Browserweichen zu erstellen?

3. Es ist ein Programm zu entwickeln, durch welches alle verfügbaren Dateiendungen, die vom Netscape Navigator im Zusammenhang mit MIME-Typen interpretiert werden können, interpretiert werden.

9 Formulare und Formular-Elemente

lernen

Ziele dieses Kapitels

In diesem Kapitel lernen Sie den Umgang mit Formularen und den hierin enthaltenen Elementen. Bei diesen Elementen handelt es sich beispielsweise um Eingabefelder und Auswahllisten. Obwohl Formulare durch einfache HTML-Syntax erzeugt werden können, kommt deren wahre Stärke erst durch eine Symbiose mit JavaScript zur Geltung. Hierdurch lassen sich u.a. Nutzereingaben überprüfen, Listenfelder mit neuen Einträgen belegen und Checkboxen manipulieren.

9.1 Zugriff auf Formulare

Es wird auf das gesamte Formular zugegriffen.

Um Formulare einer HTML-Datei ansprechen zu können, wird das **forms**-Objekt verwendet. Beachten Sie, dass hierdurch auf das gesamte Formular, nicht aber auf einzelne Formularelemente zugegriffen wird. Wie Sie auf einzelne Elemente zugreifen können, wird im Laufe dieses Kapitels noch gezeigt. Vor einem solchen Zugriff ist darauf zu achten, dass innerhalb der HTML-Datei ein Formular auch tatsächlich durch das **<FORM>**-Tag ausgezeichnet wird. Es stehen drei verschiedene Zugriffsvarianten für Formulare zur Auswahl.

```
document.forms[#].Eigenschaft
document.forms[#].Methode()
```

Um mit Hilfe einer Indexnummer auf ein Formular zuzugreifen, wird zunächst die Anweisung **document.forms** notiert. Innerhalb der eckigen Klammern wird eine Zahl angegeben. Diese Zahl beschreibt, das wievielte Formular innerhalb der Datei angesprochen werden soll. Die Zählung beginnt bei *0*. Soll auf das 3. Formular zugegriffen werden, muss in den eckigen Klammern die Zahl *4* angegeben werden. Zwei Beispiele:

```
document.forms[1].length
document.forms[3].reset()
```

Eine weitere Zugriffsmöglichkeit besteht in der Verwendung von Indexnamen. Auch hierbei wird die Anweisung **document.forms** angegeben. Innerhalb der geschweiften Klammern wird der Name des Formulars in Anführungszeichen gesetzt notiert. Abschließend wird die Eigenschaft oder Methode angegeben.

```
document.forms["Formularname"].Eigenschaft
document.forms["Formularname"].Methode()
```

Im folgenden Beispiel wird somit auf das Formular `Formular` zugegriffen. Der Indexname, mit dem der Zugriff auf das Formular erfolgt, ergibt sich aus dem Wert des **name**-Attributs im einleitenden **<FORM>**-Tag. Hinter der schließenden eckigen Klammer werden in dem Beispiel die **action**-Eigenschaft sowie die **submit()**-Methode notiert.

```
document.forms["Formular"].action
document.forms["Formular"].submit()
```

Die dritte Möglichkeit für den Zugriff auf Formulare besteht darin, den Namen des Formulars direkt anzugeben. Dieser Name ergibt sich auch hierbei aus dem Wert, der dem **name**-Attribut des einleitenden **<FORM>**-Tags zugewiesen wurde. Hinter dem **document**-Objekt wird dieser Name angegeben. Hieran schließt sich die Notation der Eigenschaft bzw. Methode an.

```
document.Formularname.Eigenschaft
document.Formularname.Methode()
```

Im folgenden Beispiel wird auf das Formular `Formular` zugegriffen. Als Eigenschaft wird **target** und als Methode **reset()** angewandt. Mehr zu den möglichen Eigenschaften und Methoden des **forms**-Objekts erfahren Sie in diesem Kapitel.

```
document.Formular.target
document.Formular.reset()
```

Beachten Sie, dass der Formularname innerhalb des Script-Bereichs exakt in der gleichen Schreibweise wie beim **name**-Attribut notiert werden muss. JavaScript unterscheidet zwischen Groß- und Kleinschreibung.

9.1.1 Wohin wird das Formular geschickt?

Wert des* action*-Attributs

Anhand der **action**-Eigenschaft kann der Wert, der innerhalb des **action**-Attributs im einleitenden **<FORM>**-Tag definiert wurde, ausgelesen werden. Bei diesem angegeben Wert kann es sich beispielsweise um eine E-Mail-Adresse, ein CGI- oder CFML-Script handeln. Im folgenden Beispiel wird die **action**-Eigenschaft dazu verwendet, um dem Anwender, der das Formular ausgefüllt hat, vor dessen Absenden nochmals anzei-

gen zu können, wohin die Formulardaten gesandt werden. Der Anwender kann nach dieser Anzeige selbst entscheiden, ob die Daten tatsächlich versandt werden sollen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Email()
{
var Versand = window.confirm("Dieses Formular geht an " +
document.Formular.action);
return Versand;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action=mailto:mail@myserver.com
onsubmit="return Email()">
<INPUT  type="text" size="20" name="E-Mail">
<INPUT  type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.1: Hinweis auf den Adressaten des Formulars

Mit dem Absenden des Formulars wird die Funktion `Email()` ausgelöst. Bei deren Aufruf wird ein modulares Standardfenster geöffnet. In diesem wird ein Hinweistext ausgegeben, welcher dem Anwender den genauen Zielort, in diesem Fall also eine E-Mail-Adresse, der Formulardaten bekannt gibt. Bestätigt der Nutzer die innerhalb des modularen Fensters angezeigte OK-Schaltfläche, werden die eingegeben Daten versendet. Beim Anklicken des Abbrechen-Buttons wird der Sendevorgang indes abgebrochen. Aber auch in diesem Fall bleiben die eingegebenen Daten, so es sich denn der Anwender noch anders überlegen würde, erhalten.

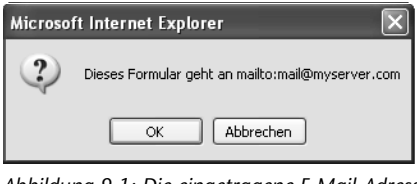


Abbildung 9.1: Die eingetragene E-Mail-Adresse wird angezeigt.

9.1.2 Formatierungsart bestimmen

Wert des **encoding**-*Attributs*

Im Zusammenhang mit dem Versenden von Formulardaten kann das innerhalb des einleitenden **<FORM>**-Tags notierte **enctype**-Attribut von entscheidender Bedeutung sein. Dies gilt vor allem im Zusammenhang mit dem Verschicken von Daten an eine E-Mail-Adresse. Denn nur wenn in diesem Fall die Angabe **enctype="text/plain"** notiert wird, bekommt der Empfänger ordentlich formatierte Daten. Per JavaScript kann der Wert, welcher dem **encoding**-Attribut zugewiesen wurde, über die Eigenschaft **encoding** ausgelesen werden. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Formatierung()
{
alert(document.Formular.encoding)
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action=mailto:mail@myserver.com
onsubmit="Formatierung()" enctype="text/plain">
<INPUT  type="text" size="20" name="E-Mail">
<INPUT  type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.2: Der Wert des **enctype**-*Attributs wird in einem Meldungsfenster ausgegeben.*

Innerhalb des gezeigten Beispiels werden die vom Anwender notierten Daten über die **mailto**-Methode an eine E-Mail-Adresse gesandt. Um die zu empfangenen Daten in einer lesbaren Art und Weise formatiert zu erhalten, wurde innerhalb des einleitenden **<FORM>**-Tags das Attribut **enctype** mit dem Wert **text/plain** notiert. Beim Absenden des Formulars wird durch den Event-Handler **onsubmit** die Funktion **Formatierung()** aufgerufen und dem Anwender innerhalb eines **alert()**-Fensters die verwendete Kodierung angezeigt.

9.1.3 Anzahl aller Formulare

<FORM>-*Tag korrekt anwenden*

In einigen Fällen kann die Ermittlung der Gesamtanzahl von Formularen innerhalb einer Datei sinnvoll sein. Für diesen Zweck steht die Eigenschaft **length** zur Verfügung. Um die **length**-Eigenschaft verwenden

zu können, müssen die innerhalb der entsprechenden Datei notierten Formulare durch das **<FORM>**-Tag korrekt eingerahmt sein.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Formular1" action="mailto:myname@myserver.de">
<INPUT  type="text" value="">
<INPUT  type="submit" value="senden">
</FORM>
<FORM name="Formular " action="mailto:myname@myserver.de">
<INPUT  type="text" value="">
<INPUT  type="submit" value="senden">
</FORM>
<FORM name="Formular " action="mailto:myname@myserver.de">
<INPUT  type="text" value="">
<INPUT  type="submit" value="senden">
</FORM>
<SCRIPT language="JavaScript" type="text/javascript">
<!--
document.write('Bitte füllen Sie alle '+ document.forms.length + " Formulare
aus.");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 9.3: Unterhalb aller Formulare wird deren Anzahl angezeigt.

Die aufgeführte Syntax hat die Darstellung von drei Formularen zur Folge. Über die **document.write()**-Methode wird unter den drei Formularen dynamisch ein Text generiert, in dem die Gesamtanzahl der integrierten Formulare ausgegeben wird. Diese Syntaxform ist immer dann von Vorteil, wenn ein entsprechender Hinweistext, der den Anwender auf die Gesamtanzahl der Formulare hinweist, angezeigt werden soll, ohne dass dieser Text permanent per Hand neu geniert werden muss.

9.1.4 Versandmethode

Wert des **method**-*Attributs*

Die Versandmethode von Formularen über das HTML-Attribut **method** im einleitenden **<FORM>**-Tag lässt sich per JavaScript anhand der **method**-Eigenschaft auslesen. Die Bestimmung der Versandmethode orientiert sich vor allem an der gewünschten Verarbeitung der zugesandten Formulardaten. Sollen die Daten per E-Mail verschickt werden, muss die Methode **post** notiert sein. Im Zusammenhang mit Scripts für die Verar-

beitung der ankommenden Informationen wird indes die **get**-Methode eingesetzt. Praktische Anwendungen der JavaScript-Eigenschaft **method** lassen sich auf vielfältige Weise realisieren, wobei hierfür die folgende Syntax exemplarisch einen Einblick liefern soll:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Methode() {
var formular = document.formular;
 if(formular.action.indexOf("@") > 0)
  formular.method = "post";
 else
  formular.method = "get";
 return true;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="formular" action=mailto:ich@deins.de
onsubmit="return Methode()">
Name: <INPUT  type="text" size="40" name="name">
<INPUT  type="submit" value="Absenden">
</FORM>
</BODY>
</HTML>
```

Listing 9.4: Die Funktion überprüft, ob das @-Zeichen in dem Eingabefeld vorkommt.

Mit dem Absenden des Formulars wird die Funktion `Methode()` ausgeführt. Anhand dieser Funktion wird überprüft, ob innerhalb des einzeiligen Eingabefeldes das **@**-Zeichen vorkommt. Da dieses Zeichen als untrüglicher Hinweis auf eine E-Mail-Adresse gewertet werden kann, wird das Formular mit der **post**-Methode versandt. Anderenfalls, kommt also innerhalb des Eingabefeldes kein **@**-Zeichen vor, wird die **get**-Methode verwendet. Zwei Faktoren spielen in der gezeigten Funktion eine entscheidende Rolle. Zunächst wird anhand der **indexOff()**-Methode die innerhalb des Eingabefeldes vorkommende Zeichenkette nach einem eventuell vorkommenden **@**-Zeichen durchsucht. Zum anderen erwartet der Event-Handler **onsubmit** in diesem Beispiel **true** als Rückgabewert.

9.1.5 Formularname

Wert des **name***-Attributs*

Der Name des Formulars in einer HTML-Datei kann über die **name**-Eigenschaft ausgelesen werden. Dieser Name kann vor allem dann von Bedeutung sein, wenn ein direkter Zugriff auf ein Formular realisiert oder aber Informationen über dieses angezeigt werden sollen. Das folgende Beispiel beschreibt eine simple **name**-Anwendung im Zusammenspiel mit dem Event-Handler **onsubmit**.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Email" action="mailto:maske@gruen.de"
onsubmit="Formularname()">
E-Mail: <INPUT  type="text" size="40" name="email">
<INPUT  type="submit" value="Absenden">
</FORM>
<SCRIPT type="text/JavaScript">
<!--
function Formularname()
{
 alert("Sie senden jetzt das Formular " + document.Email.name);
}
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 9.5: Die Ziel-Adresse wird in einem Meldungsfenster ausgegeben.

Mit dem Absenden des Formulars wird innerhalb eines **alert()**-Fensters der Name des Formulars, in diesem Beispiel ist dies `EMail`, ausgegeben. Eine solche Anwendung erscheint vor allem dann sinnvoll, wenn innerhalb einer HTML-Seite mehrere voneinander unabhängige Formulare definiert wurden. Somit kann der Anwender vor dem Absenden der Formulardaten nochmals explizit darauf hingewiesen werden, welches Formular abgesendet wird.

9.1.6 Zielfenster für Serverantworten

Feedback-Formulare realisieren

Gerade im Zusammenhang mit einer serverseitigen Verarbeitung von Formularinhalten ist die Wahl des Zielfensters für eine Rückmeldung an den Anwender wichtig. So ist es mittlerweile gute Sitte, dass die Person, welche ein Feedback-Formular ausfüllt, umgehend eine Bestätigung bekommt, sobald die Daten korrekt übertragen wurden. Problematisch ist

eine solche serverseitige Antwort allerdings im Zusammenhang mit Frames und geöffneten Popup-Fenstern. Standardmäßig wird eine solch serverseitige Antwort in dem gleichen Fenster geöffnet, von dem aus auch die Anfrage gesendet wurde. Dass dies nicht in jedem Fall und in dieser Form gewollt ist, steht freilich außer Frage. Um in solchen Fällen eine gezielte Zielfensterbasis spezifizieren zu können, steht die **target**-Eigenschaft zur Verfügung. Die **target**-Eigenschaft erwartet als Wert einen Fensternamen entweder eines existierenden Framesets oder eines geöffneten Popup-Fensters.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Bestaetigung() {
 document.Formular.target = "DankeFenster";
 return true;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="include.cfm"
onsubmit="return Bestaetigung()">
<INPUT  type="text" size="40" name="Eingabe">
<INPUT  type="submit" value="Absenden">
</FORM>
</BODY>
</HTML>
```

Listing 9.6: Es wird nach dem Absenden des Formulars eine Danke-Seite angezeigt.

Zur Realisierung des gezeigten Beispiels muss angenommen werden, dass sich die oben aufgeführte HTML-Datei innerhalb eines Framesets befindet. Parallel zu dieser Datei existiert innerhalb dieses Framesets noch ein weiterer Frame mit dem Namen `DankeFenster`. Durch den Event-Handler **onsubmit** wird die Funktion `Bestaetigung()` aufgerufen. Diese Funktion sendet die Formulardaten an ein CFM-Script. Das Script schickt wiederum ein Dankesschreiben an den Anwender zurück, welches durch die Befehlsfolge **document.formular.target =**`"DankeFenster"` innerhalb des Frames `DankeFenster` angezeigt wird.

9.1.7 Inhalte löschen

gleiche Funktion wie ein **Reset**-*Button*

Die **reset()**-Methode in JavaScript hat auf die Formulardaten die gleiche Wirkung wie ein **Reset**-Button in HTML: Alle Eingaben innerhalb des spezifizierten Formulars werden gelöscht. Anders als durch einen HTML-Standard-Button kann das Löschen der Formularinhalte per JavaScript jedoch kontrolliert und somit ein versehentliches Anklicken des Löschen-Buttons verhindert werden. Ein Beispiel:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function loescheInhalt()
{
Bestaetigung = confirm("Wollen Sie wirklich alle Eingaben
löschen?");
if (Bestaetigung == true) document.Formular.reset();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="mailto:server@myserver.de">
<INPUT  type="text" size="20" name="Eingabe">
<INPUT  type="submit" value="Absenden">
<INPUT  type="button" value="löschen"onclick="loescheInhalt()">
</FORM>
</BODY>
</HTML>
```

Listing 9.7: Vor dem Löschen der Formular-Daten wird nochmals nachgefragt.

Innerhalb der gezeigten Syntax wurde ein Formular mit dem Namen `Formular` definiert. Dieses enthält ein einzeiliges Eingabefeld sowie je einen Button zum Absenden und zum Löschen der Formularinhalte. Während der **submit**-Button durch Standard-HTML dargestellt wird, ist der **Reset**-Button per JavaScript gesteuert. Die Funktion `loescheInhalt()`, die durch den Event-Handler **onclick** ausgelöst wird, öffnet nach dem Anklicken des **Reset**-Buttons ein Dialogfenster, in welchem der Anwender darauf hingewiesen wird, dass er im Begriff steht, alle Formulardaten zu löschen. Daraufhin hat der Nutzer zwei Möglichkeiten zur weiteren Verfahrensweise: Bestätigt er die Abfrage, klickt er also den OK-Button, werden alle Daten gelöscht. War das Auslösen des **Reset**-Buttons jedoch nur ein Versehen und wird dieses durch das Anklicken des Abbrechen-Buttons bestätigt, bleiben dem Anwender seine eingegeben Daten innerhalb des Formulars erhalten.

9.1.8 Formulare senden

gleiche Funktion wie ein **Submit**-*Button*

Von ihrer Wirkung her identisch mit einem **Submit**-Standard-Button in HTML ist die JavaScript-Methode **submit()**. Auch diese Methode hat das Absenden der Formulardaten zur Folge. Im Unterschied zu herkömmlicher HTML-Syntax lässt sich ein über **submit()** realisierter Button jedoch kontrollieren und somit weitaus vielfältiger verwenden. Bislang problematisch war die Verwendung der **submit()**-Methode im Zusammenhang mit einigen Versionen des Netscape Navigators. Zwar interpretierte dieser in den Produktversionen 2.0 und 3.0 **submit()** korrekt, dies änderte sich jedoch mit der Einführung der Versionen 4.x. Diese Browser können die **submit()**-Methode tatsächlich nur noch dann verarbeiten, wenn innerhalb des einleitenden **<FORM>**-Tags dem Attribut **action** ein Script-Aufruf zugewiesen wurde. Bei der Verwendung einer E-Mail-Adresse wird **submit()** indes geflissentlich ignoriert. Diese Netscape-Fehlentwicklung wurde mit der Produktversion 6.0 des Navigators jedoch wieder korrigiert. Die folgende Syntax beschreibt, wie sich das Absenden des Formulars überprüfen lässt und dem Anwender nochmals die Möglichkeit geboten wird, über seinen Wunsch, nämlich das Absenden des Formulars, nachdenken zu können.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function sendeFormular()
{
Bestaetigung = confirm("Ihre Daten sind nicht verschlüsselt \n Trotzdem
senden?");
if (Bestaetigung == true) document.Formular.submit();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="mailto:server@myserver.de">
<INPUT  type="text" size="20" name="Eingabe">
<INPUT  type="button" value="Absenden"
onclick="sendeFormular()">
<INPUT  type="reset" value="löschen">
</FORM>
</BODY>
</HTML>
```

Listing 9.8: Vor dem Absenden wird nochmals gefragt, ob das Formular tatsächlich abgeschickt werden soll.

Die gezeigte Syntax hat zur Folge, dass durch das Klicken auf den Absende-Button die Funktion `sendeFormular()` aufgerufen wird. Diese öffnet ein Dialogfenster, in welchem der Anwender nochmals gefragt wird, ob die Daten, trotz ihrer Nicht-Verschlüsselung, abgesendet werden sollen. Wird dies durch einen Klick auf den Abbrechen-Button verneint, werden die Daten nicht versendet. Das gegenteilige Ereignis, also das Anklicken des OK-Buttons, führt indes zum Absenden der Formulardaten.

9.2 Auf Formularelemente zugreifen

nicht auf alle Elemente anwendbar

Durch das **elements**-Objekt können Sie auf Elemente eines Formulars zugreifen. Beachten Sie, dass Sie hierdurch allerdings nicht Zugriff auf alle Formularelemente haben. Auswahllisten können Sie mit diesem Objekt nicht ansprechen. Hierfür existiert das **options**-Objekt. Auf dieses Objekt wird im Abschnitt 9.3 ausführlich eingegangen. Mit dem **elements**-Objekt können Sie auf die in der folgenden Tabelle aufgeführten Formularelemente zugreifen.

Element	Beschreibung
<INPUT type="button">	Klick-Button
<INPUT type="checkbox">	Checkbox
<INPUT type="file">	Datei-Upload
<INPUT type="hidden">	verstecktes Eingabefeld
<INPUT type="password">	Passwort-Feld
<INPUT type="radio">	Radio-Button
<INPUT type="reset">	Löschen-Button
<INPUT type="submit">	Senden-Button
<INPUT type="text">	einzeiliges Eingabefeld
<TEXTAREA></TEXTAREA>	mehrzeiliges Eingabefeld

Tabelle 9.1: Zulässige HTML-Elemente

Auch beim **elements**-Objekt existieren wieder verschiedene Varianten für den Zugriff. So können Sie beispielsweise über eine Indexnummer auf das gewünschte Formularelement zugreifen. Hierfür notieren Sie die Anweisung **document.forms**. In eckigen Klammern wird angegeben, das wievielte Formular der Datei angesprochen werden soll. Durch einen Punkt getrennt wird das **elements**-Objekt notiert. Innerhalb der eckigen Klammern geben Sie durch eine Ziffer an, auf das wievielte Element des Formulars zugegriffen werden soll. Abschließend wird die Eigenschaft oder Methode notiert.

```
document.forms[#].elements[#].Eigenschaft
document.forms[#].elements[#].Methode()
```

Die beiden folgenden Syntaxformen greifen auf zwei unterschiedliche Formulare einer Datei zu. Im ersten wird das zweite und im zweiten das dritte Formular angesprochen. Beachten Sie, dass die Zählung bei `0` beginnt. Das erste Formular einer Datei wird demnach mit `0` angesprochen. Beide Varianten greifen auf das erste Formularelement zu.

```
document.forms[1].elements[0].checked = true;
document.forms[2].elements[0].click();
```

Eine weitere Möglichkeit besteht darin, das Element über seinen Namen anzusprechen. Hierfür wird hinter dem **`document`**-Objekt zunächst der Formularname notiert. Hierbei muss exakt der gleiche Name wie im einleitenden **`<FORM>`**-Tag beim **`name`**-Attribut angegeben werden. Gleiches gilt für den sich hieran anschließenden Elementnamen. Abschließend wird die Eigenschaft oder Methode notiert.

```
document.Formularname.Elementname.Eigenschaft
document.FormularName.Elementname.Methode()
```

Durch die folgende Syntax wird das Formular `Formular` angesprochen. Hierin befindet sich ein Formularelement mit dem Namen `Eingabe`, auf das nun zugegriffen und eine Eigenschaft bzw. Methode angewandt werden kann.

```
document.Formular.Eingabe.name
document.Formular.Eingabe.blur()
```

Checkboxen und Radio-Buttons als Sonderfall

Einen Sonderfall beim Zugriff auf Formularelemente müssen wir noch betrachten. Bei Checkboxen und Radio-Buttons handelt es sich im Normalfall um gruppierte Elemente. Durch eine Gruppierung wird erreicht, dass jeweils nur eines dieser Elemente markiert werden kann. Eine typische Anwendung für solche Gruppen ist beispielsweise die Frage, ob es sich bei dem Anwender um einen Mann oder eine Frau handelt. Um eine solche Gruppierung zu erreichen, werden Checkboxen oder Radio-Buttons die gleichen Werte beim **`name`**-Attribut zugewiesen. Eine Unterscheidung der einzelnen Elemente findet ausschließlich durch die unterschiedliche Wertzuweisung beim **`value`**-Attribut statt. Ein Beispiel:

```
<FORM name="Formular">
<INPUT type="radio" name="Autoren" value="Hornby">
<INPUT type="radio" name="Autoren" value="Ellis">
<INPUT type="radio" name="Autoren" value="Roth">
</FORM>
```

In dem Formular `Formular` befinden sich drei Radio-Buttons. Allen dreien wird der gleiche Name `Autoren` zugewiesen. Hierdurch wird erreicht, dass immer nur einer dieser Buttons selektiert werden kann. Die Unterscheidung wird über die jeweils anderen Werte des **`value`**-Attributs erreicht.

Wie kann man nun per JavaScript auf eine solche Gruppe zugreifen? Nehmen wir an, wir wollen durch JavaScript den zweiten Radio-Button automatisch selektieren.

```
document.Formular.Autoren[1].checked = true
```

Gruppen sind Arrays

Gruppen werden in JavaScript als Arrays behandelt. Um hierauf zuzugreifen, muss zunächst das **document**-Objekt notiert und anschließend das Formular angegeben werden. Hieran schließt sich der Name der Gruppe an. In der eckigen Klammer wird die Indexnummer des Elements angegeben. Beachten Sie hierbei, dass die Zählung bei `0` beginnt. Da wir in diesem Beispiel auf den zweiten Radio-Button zugreifen wollen, wird hier der Wert *1* angegeben.

9.2.1 Status von Checkboxen und Radio-Buttons

Wurde ein Element ausgewählt?

Anhand der **checked**-Eigenschaft kann der Status von Checkboxen und Radio-Buttons überwacht werden. So lässt sich durch **checked** überprüfen, ob ein solches Element vom Anwender selektiert wurde. Eingesetzt wird die **checked**-Eigenschaft vor allem innerhalb von Umfrageerhebungen mittels Formularen. Gleichfalls ist es aber auch möglich, direkt auf Nutzereingaben zu reagieren. Die **checked**-Eigenschaft kennt die beiden folgenden Werte:

mögliche Werte

- **true** – ausgewählt
- **false** – nicht ausgewählt

Sie sollten die **checked**-Eigenschaft bei Radio-Buttons nur in Verbindung mit einer Elementgruppierung einsetzen. Hierdurch wird garantiert, dass nur jeweils ein Element selektiert werden kann. Um diesen Effekt zu erreichen, muss den Radio-Buttons die zu einer Gruppe zusammengefasst werden sollen, das **name**-Attribut zugewiesen werden. Diesem muss bei allen Radio-Buttons der gleiche Wert gegeben werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Hintergrundfarbe()
{
 if(document.Formular.Farbe[0].checked == true)
  document.all.hinten.style.backgroundColor = "yellow";
 else if(document.Formular.Farbe[1].checked == true)
  document.all.hinten.style.backgroundColor = "green";
 else
  document.all.hinten.style.backgroundColor = "blue";
}
```

```
//-->
</SCRIPT>
</HEAD>
<BODY id="hinten">
<FORM name="Formular">
<INPUT  type="radio" name="Farbe" value="gelb">Gelb
<INPUT  type="radio" name="Farbe" value="gr&uuml;n">Gr&uuml;n
<BR>
<A href="javascript:Hintergrundfarbe()">W&auml;hlen Sie eine
Hintergrundfarbe</A>
</FORM>
</BODY>
</HTML>
```

Listing 9.9: Die Hintergrundfarbe kann verändert werden.

Das Beispiel enthält ein Formular, in welchem eine Gruppe von zwei Radio-Buttons definiert ist. Diese Gruppenbildung wird erreicht, indem beiden Elementen das Attribut **name** mit dem dazugehörenden Wert `Farbe` zugewiesen wird. Durch Anklicken des Hyperlinks wird die `Hintergrundfarbe()`-Funktion aufgerufen. Hierin wird in Form einer **if**-Abfrage überprüft, ob ein und, wenn ja, welcher Radio-Buttons selektiert ist. Handelt es sich hierbei um den ersten Radio-Button, wird die Hintergrundfarbe des Dokuments auf Gelb gesetzt. Wurde hingegen der zweite ausgewählt, wird Grün als Hintergrundfarbe angewandt. Wird indes kein Radio-Button selektiert, erfolgt die Anzeige einer blauen Hintergrundfarbe.

9.2.2 Voreingestellter Wert von Checkboxen und Radio-Buttons

In HTML lassen sich Radio-Buttons und Checkboxen durch den Einsatz des **checked**-Attributs als vorselektiert definieren. Das betreffende Element wird also per Vordefinition, und ohne dass der Anwender hierauf Einfluss nimmt, als markiert dargestellt. Geeignet ist dies vor allem immer dann, wenn dem Anwender ein Vorschlag unterbreitet wird, welches Element denn nun von ihm am ehesten selektiert werden sollte. Dieses Verhalten lässt sich durch den Einsatz der **defaultChecked**-Eigenschaft überprüfen.

mögliche Werte

- **true** – ausgewählt
- **false** – nicht ausgewählt

Im folgenden Beispiel befinden sich zwei Radio-Buttons, von denen einer durch den Einsatz der **checked**-Eigenschaft als selektiert definiert ist. Dem Anwender wird nach dem Anklicken des Formular-Buttons ein

dynamisch generierter Text angezeigt. Dies gilt jedoch nur dann, wenn der Browser die **checked-** bzw. **defaultChecked**-Eigenschaft korrekt interpretiert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Pruefer() {
 if(document.formular.einhalt[0].defaultChecked == true)
  document.write("Sie haben richtig gew&auml;hlt");
 else
  alert("So geht es nun aber nicht");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="formular">
<INPUT  type="radio" name="einhalt" value="ja" checked>
<INPUT  type="radio" name="einhalt" value="nein">
<BR><INPUT  type="button" value="Starten" onclick="Pruefer()">
</FORM>
</BODY>
</HTML>
```

Listing 9.10: Die Richtigkeit der Auswahl wird überprüft.

Mit dem Anklicken des Formular-Buttons wird die `Pruefer()`-Funktion ausgelöst. Diese überprüft in Form einer **if**-Abfrage, ob der Wert der **defaultChecked**-Eigenschaft auf **true** gesetzt wurde. Ist dies der Fall, wird ein entsprechender Text dynamisch in das Dokument geschrieben. Ist der Wert indes **false**, wird über die **alert()**-Methode ein Hinweistext in einem Meldungsfenster ausgegeben.

9.2.3 Voreingestellter Wert von Eingabefeldern

Tippzeit mindern

Über das **value**-Attribut können einzeilige Texteingabefelder mit einem Wert vorbelegt werden. Dies ist immer dann von Nutzen, wenn dem Anwender wertvolle Tippzeit erspart werden soll. Der hier angegebene Text kann anhand der **defaultValue**-Eigenschaft gespeichert und verändert werden. Sie sollten bei der Vorbelegung eines Textfeldes stets darauf achten, dass hier, ebenso wie in normalem Fließtexten, Umlaute gesondert gekennzeichnet werden. Das folgende Beispiel beschreibt eine Anwendung, mit deren Hilfe zweierlei nutzerrelevante Sachverhalte realisiert werden können. Dem Anwender wird zunächst innerhalb eines

einzeiligen Eingabefeldes ein URL angezeigt. Durch Anklicken des Formular-Buttons wird dieses Verweisziel angezeigt. Zusätzlich wird bei der Anzeige des ersten URL überprüft, welche Sprache der Browser des Anwenders hat. Ist diese Englisch, wird der URL einer englischsprachigen Seite angezeigt. Anderenfalls erscheint innerhalb des Eingabefeldes ein URL mit der Top-Level-Domain *de*.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeURL()
{
window.location.href = document.Formular.URL.value;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
Ihre Adresse: <INPUT  name="URL" value="http://www.myhost.de/">
<INPUT  type="button" value="los" onclick="zeigeURL()">
</FORM>
<SCRIPT type="text/JavaScript">
<!--
if(navigator.userAgent.indexOf("en") > 0)
{
document.Formular.URL.defaultValue = "http://www.myhost.com/";
document.Formular.URL.value =document.Testform.url.defaultValue;
}
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 9.11: Der URL wird je nach verwendeter Sprachversion des Browsers angezeigt.

Das Eingabefeld `URL` wurde zunächst mit dem Text `http://www.myhost.de` vorbelegt. Beim Einlesen der Datei wird der unterhalb des Eingabefeldes notierte Script-Bereich ausgeführt. In diesem wird die Sprachversion des Browsers überprüft. Dies geschieht anhand der **`userAgent`**-Eigenschaft des **`navigators`**-Objekts. Durch die **`indexOff()`**-Methode wird überprüft, ob hierin der Wert `en` vorhanden ist. Ist dies der Fall, wird der voreingestellte Wert des Eingabefeldes in `http://www.myhost.com` geändert. Durch das Anklicken des Formular-Buttons wird der URL, welcher innerhalb des Eingabefeldes aktuell steht, über **`location.href`** geöffnet.

Abbildung 9.2: Es wurde ein deutschsprachiger Browser verwendet.

9.2.4 Das aktuelle Formular ermitteln

Das **<FORM>***-Tag nicht vergessen!*

Im Zusammenhang mit der Verwendung von Formularelementen ist darauf zu achten, dass diese stets innerhalb eines **<FORM>**-Tags stehen sollten. Dies gilt auch dann, wenn diese Elemente nicht zur Übermittlung von Daten, sondern für andere Zwecke eingesetzt werden. Denn zum einen dürfen Formularelemente ausschließlich innerhalb eines entsprechenden Formulars vorkommen, andererseits ist dieses Vorgehen aber gerade im Hinblick auf den Netscape Navigator notwendig. Dieser stellt Formularelemente nämlich nur dann dar, wenn sich diese innerhalb eines **<FORM>**-Tags befinden. Über die **form**-Eigenschaft kann auf das betreffende Formular zugegriffen werden. Beachten Sie, dass die alleinige Verwendung der **form**-Eigenschaft nicht zu dem gewünschten Ergebnis führt. In diesem Fall wird lediglich das **form**-Objekt zurückgeliefert. Um detailliertere Informationen über das eingesetzte Formular zu erhalten, sollte eine Eigenschaft oder Methode des **forms**-Objekts, wie beispielsweise **name,** notiert werden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT  type="button" name="Email" value="Formularname"
onclick="alert(document.Formular.Email.form.name)">
</FORM>
</BODY>
</HTML>
```

Listing 9.12: Der Name des Formulars wird in einem Meldungsfenster ausgegeben.

Dem Formular wird hier der Name `Formular` zugewiesen. Das **<FORM>**-Tag enthält einen Formular-Button. Dessen Aktivierung gibt innerhalb eines Meldungsfensters den Namen des Formulars aus, in welchem sich der Button befindet. Hierzu wird über **document.**`Formular.Email`**.form** auf das Formularelement zugegriffen. Um den Formularnamen ausgeben zu können, wird hier die **name**-Eigenschaft des **forms**-Objekts verwendet.

9.2.5 Namen von Formularelementen ermitteln

keine eindeutige Identifikation

Über die **name**-Eigenschaft kann der Name eines Formularelements ausgelesen werden. Bezug wird hierbei auf das innerhalb der Elementdefinition notierte HTML-Attribut **name** genommen. Sie sollten berücksichtigen, dass durch die Verwendung des **name**-Attributs ein Formularelement nicht eindeutig spezifiziert wird. Innerhalb eines Formulars ist die Notation mehrerer Elemente möglich, bei denen dem **name**-Attribut der gleiche Wert zugewiesen wird. Als typisches Beispiel sei hier eine Gruppe von Radio-Buttons genannt. Diese kann vom Browser nur dann als Elementgruppe erkannt werden, wenn dem **name**-Attribut der jeweils gleiche Wert zugewiesen wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Nameaendern()
{
document.Formular.Eingabe.value =
document.Formular.neuerWert.name;
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT  type="text" name="Eingabe" value="Test">
<INPUT  type="submit" name="neuerWert" value="abschicken">
<A href="javascript:Nameaendern()">Der Name des Buttons</A>
</FORM>
</BODY>
</HTML>
```

Listing 9.13: Der Name des Formular-Buttons wird im Eingabefeld angezeigt.

Das Formular enthält ein einzeiliges Eingabefeld sowie einen Formular-Button. Durch dessen Anklicken wird die Funktion `Nameaendern()` aufgerufen. Mittels dieser ist es möglich, den Namen des Buttons innerhalb des Eingabefeldes anzeigen zu lassen. Hierzu wird der Wert dieses Eingabefeldes mit dem Namen des Formular-Buttons gleichgesetzt. Nach dem Anklicken des Buttons wird der vordefinierte Wert des Eingabefelds durch den Wert `neuerWert` ersetzt.

9.2.6 Typ von Formularelementen ermitteln

Es wird auf das **type***-Attribut Bezug genommen.*

Von welchem Typ ein Formularelement ist, lässt sich anhand der **type**-Eigenschaft ermitteln. Dies ist vor allem dann von Bedeutung, wenn man sich vor Augen führt, dass für die Darstellung von unterschiedlichen Formularelementen zwar das gleiche HTML-Tag verwendet wird, die eigentliche Typunterscheidung jedoch durch die Zuweisung des **type**-Attributs und eines entsprechenden Wertes realisiert wird. So wird beispielsweise über **type="text"** ein einzeiliges Eingabefeld erzeugt, wohingegen über **type="password"** ein Passwortfeld generiert wird. Sie sehen also: Das gleiche Attribut, jedoch ein anderer Wert haben hierbei die Darstellung zweier gänzlich unterschiedlicher Formularelemente zur Folge. Die **type**-Eigenschaft liest den Wert des **type**-Attributs aus. Das nachstehende Beispiel enthält unter anderem ein Eingabefeld. Durch dessen Überfahren mit der Maus wird innerhalb eines Meldungsfensters dessen Typ, hier also *text*, angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Typerkennung()
{
  alert(document.Formular.URL.type);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<h2>Kleine HTML-Kunde</h2>
<FORM name="Formular">
<INPUT  type="text" name="URL" value="was f&uuml;r ein Feld"
onmouseover="Typerkennung()">
</FORM>
</BODY>
</HTML>
```

Listing 9.14: Der Typ des Eingabefeldes wird in einem Meldungsfenster angezeigt.

Um den beschriebenen Effekt zu realisieren, wird das einzeilige Eingabefeld mit dem Event-Handler **onmouseover** ausgestattet. Dieser löst die `Typerkennung()`-Funktion aus. Hierin wird zunächst über **document**`.Formular.URL` auf das Eingabefeld zugegriffen. Um den verwendeten Elementtyp zu ermitteln, wird anschließend die **type**-Eigenschaft notiert. Die letztendliche Ausgabe des Elementtyps wird über die **alert()**-Methode verwirklicht.

Abbildung 9.3: Der Typ des Eingabefeldes wird ausgegeben.

9.2.7 Werte von Formularelementen ermitteln

unterschiedliche Bedeutungen des* `value`*-Attributs

Um den Wert eines Formularelements zu ermitteln, wird die **`value`**-Eigenschaft eingesetzt. Hierbei spielt es keine Rolle, ob dieser Wert per Voreinstellung oder erst durch eine späterere Eingabe zugewiesen wird. Für die Werte-Ermittlung greift die **`value`**-Eigenschaft auf das **`value`**-Attribut des Formularelements zu und liest dessen Wert aus. Die Behandlung des **`value`**-Attributs stellt sich in HTML recht unterschiedlich dar. So wird dieses beispielsweise im Umgang mit Formular-Button dazu benutzt, um einen Beschriftungstext zu realisieren. Bei einzeiligen Eingabefeldern kann indes eine Feldvorbelegung geschaffen werden. Wiederum bei anderen Elementen, wie beispielsweise Radio-Buttons, hat die **`value`**-Zuweisung keine sichtbaren Folgen. Im folgenden Beispiel befinden sich neben zwei Radio-Buttons ein Formular-Button sowie ein einzeiliges Eingabefeld. Innerhalb dieses Feldes wird der Wert des selektierten Radio-Buttons angezeigt. Hierfür wird allerdings nicht der voreingestellte Wert der Radio-Buttons verwendet. Vielmehr wird beiden ein neuer Wert zugewiesen. Dieser Wert wird dem Anwender angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Geschlecht()
{
if (document.Formular.Inhalt[0].checked == true)
document.Formular.Anzeige.value = "Mann";
else
document.Formular.Anzeige.value = "Frau";
}
```

```
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT  name="Anzeige" value="Ihr Geschlecht">
Mann: <INPUT  type="radio" name="Inhalt" value="0">
Frau: <INPUT  type="radio" name="Inhalt" value="1">
<INPUT  type="button" onclick="Geschlecht()" value="Auswahl">
</FORM>
</BODY>
</HTML>
```

Listing 9.15: Der Wert der Auswahl wird in dem Eingabefeld angezeigt.

Der Event-Handler **onclick,** der hier auf den Formular-Button angewandt wird, löst die `Geschlecht()`-Funktion aus. In dieser wird in Form einer **if**-Abfrage überprüft, welcher der beiden Radio-Buttons selektiert ist. Wurde der erste ausgewählt, wird der Wert der **value**-Eigenschaft von `0` auf `Mann` geändert. Für den Fall, dass der zweite Radio-Button selektiert wurde, wird dessen Wert `1` in `Frau` geändert. Der Wert `Frau` wird auch dann angezeigt, wenn keiner der beiden Radio-Buttons ausgewählt wurde.

9.2.8 Elemente fokussieren

schnellerer Zugriff auf Elemente

Die **focus()**-Methode fokussiert bzw. setzt den Cursor auf ein Formularelement. Dem Anwender wird es somit ermöglicht, ohne die Verwendung der Maus oder Tastatur in ein entsprechendes Formularfeld zu gelangen. Eingesetzt wird die **focus()**-Methode vor allem im Zusammenhang mit der Überprüfung von Nutzereingaben. So kann durch andere JavaScript-Eigenschaften die Korrektheit von Eingaben überprüft werden. Für den Fall, dass diese nicht den Entwicklerwünschen entsprechen, kann das betreffende Formularelement fokussiert werden. Der Anwender kann folglich leicht erkennen, an welcher Stelle des Formulars ihm der Fehler unterlaufen ist. Besonders im Zusammenspiel mit komplexen Formularen sollte diese Möglichkeit nicht außer Acht gelassen werden. Eine solche Anwendung beschreibt die nachfolgende Syntax. In diesem Beispiel befinden sich ein einzeiliges Eingabefeld sowie ein Formular-Button. Der Anwender wird dazu aufgefordert, innerhalb des Eingabefeldes seinen Namen anzugeben. Wird dies nicht befolgt, wird ein entsprechender Hinweistext in einem Meldungsfenster ausgegeben und das Eingabefeld anschließend fokussiert. Wurde indes ein Eintrag vorgenommen, wird dieser über die **alert()**-Methode ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function kontrolliereNamen()
{
 if(document.Formular.Vorname.value == "")
 {
  alert("Geben Sie bitte Ihren Namen an!");
  document.Formular.Vorname.focus();
 }
 else
 {
  alert("Besten Dank " + document.Formular.Vorname.value);
 }
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
Vorname<BR>
<INPUT  type=»text» name="Vorname">
<INPUT  type="button" value="senden" onclick="kontrolliereNamen()">
</FORM>
</BODY>
</HTML>
```

Listing 9.16: Der eingegebene Name wird für ein persönliches Dankeschön verwendet.

Durch anklicken des Formular-Buttons wird die Funktion `kontrolliereNamen()` ausgelöst. Diese überprüft anhand einer **`if`**-Abfrage, ob das Eingabefeld leer ist. Stellt sich diese Bedingung als wahr heraus, wird der Anwender durch einen innerhalb eines Meldungsfensters angezeigten Text aufgefordert, seinen Namen einzugeben. Bestätigt der Anwender mit dem OK-Button des Meldungsfensters, wird das Eingabefeld fokussiert. Hierfür wird zunächst über **`document.`**`Formular.Vorname` auf das Eingabefeld zugegriffen. Anschließend wird durch die Verwendung der **`focus()`**-Methode der Fokus gesetzt. Wurde indes ein Name eingetragen wird dieser, eingebettet in einen hinweisenden Text, über die **`alert()`**-Methode ausgegeben.

9.2.9 Fokus entfernen

Elemente sind schnell erreichbar.

Formularelemente können so präpariert werden, dass in ihnen der Cursor angezeigt bzw. diese fokussiert angezeigt werden. Dies ist immer dann sinnvoll, wenn der Anwender schnell zu den gewünschten For-

mularelementen gelangen und unkompliziert seine Einträge vornehmen können soll. So gut diese Möglichkeit auch ist, nicht immer ist dieser Effekt erwünscht. Durch den Einsatz der **blur()**-Methode kann der Fokus von Formularelementen entfernt werden. So kann hierdurch beispielsweise erreicht werden, dass der Anwender den Wert eines Eingabefeldes nicht verändern kann. Denkbar wäre hier folgendes Szenario. Der Anwender wählt aus einer Auswahlliste einen bestimmten Eintrag aus. Hieraufhin wird innerhalb eines Eingabefeldes ein entsprechender Wert angezeigt. Dieser soll nun durch den Anwender nicht mehr modifiziert werden können. Für eine so geartete Anwendung ist die **blur()**-Methode geradezu prädestiniert. Bei der Verwendung dieser Methode werden keine weiteren Parameter erwartet.

```
<HTML>
<HEAD>
<SCRIPT tyoe="text/javascript">
<!--
function Loeschversuch()
{
document.Formular.Eingabe.blur();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT  name="Eingabe" onfocus="Loeschversuch()" value="versuch es">
</FORM>
</BODY>
</HTML>
```

Listing 9.17: Der Inhalt des Eingabefeldes kann nicht gelöscht werden.

Durch die Fokussierung des sich innerhalb des Formulars befindlichen Eingabefeldes wird die Funktion `Loeschversuch()` ausgelöst. In dieser wird zunächst über **document.**`Formular.Eingabe` auf das einzeilige Eingabefeld zugegriffen. Durch die anschließende Notation der **blur()**-Methode wird der Fokus von dem Element entfernt. Dem Anwender wird somit nachhaltig die Möglichkeit genommen, den Wert des Eingabefeldes zu ändern. Darüber hinaus wird beim Setzen des Cursors innerhalb des Feldes dieser augenblicklich wieder entfernt.

9.2.10 Elemente automatisch anklicken

Die Methode **click()**, und dies suggeriert bereits deren Name, simuliert das Anklicken eines Elements. Problematisch kann deren Verwendung freilich immer dann sein, wenn eine Aktion ohne Wissen des Anwen-

ders ausgeführt wird. So können beispielsweise Banner angeklickt oder Formulare vorzeitig abgesendet werden. Die **click()**-Methode kann auf die folgenden Elemente angewandt werden:

Auf diese Elemente kann **click()** *angewandt werden.*

- **<INPUT type="button">** – anklickbarer Button
- **<BUTTON>** – anklickbarer Button
- **<INPUT type="file">** – Dateiupload-Button
- **<INPUT type="submit">** – Senden-Button
- **<INPUT type="reset">** – Abbruch-Button

Bedenken Sie vor dem Einsatz von **click()**, dass Anwender nur wenig erbaut darüber sein werden, dass Aktionen ohne deren Zustimmung ausgeführt werden. Auf inhaltlich bedenklichen Seiten mag diese Vorgehensweise vielleicht zum guten Ton gehören, seriöse Seitenanbieter sollten auf derlei Manipulationen allerdings verzichten. Das folgende Beispiel versucht eine nur wenig bedenkliche Anwendung zu zeigen. Hier wird dem Anwender mit dem Ausfüllen des letzten Eingabefeldes die Arbeit abgenommen, auf den Absende-Button zu klicken. Diese Aktion wird sofort mit dem Verlassen des angesprochenen Feldes ausgeführt.

```
<HTML>
<HEAD>
<SCRIPT tyoe="text/javascript">
<!--
function verschickeFormular()
{
document.Formular.senden.click();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="../cgi/pruef.pl">
Name: <INPUT  type="text" value="Ihr Name">
<BR>
Passwort: <INPUT  type="password" onblur="verschickeFormular()">
<INPUT  type="submit" name="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.18: Beim Verlassen des zweiten Eingabefeldes wird das Formular automatisch versandt.

Das Formular beinhaltet zwei einzeilige Eingabefelder sowie einen Submit-Button. Verlässt der Anwender das zweite der Eingabefelder, wird

über den Event-Handler **onblur()** die Funktion `verschickeFormular()` aufgerufen. Innerhalb dieser Funktion wird über die Anweisung **document.**`Formular.senden` das entsprechende Feld eindeutig angesprochen. Zusätzlich wird über **click()** dieses Feld automatisch angeklickt. Infolge dieses Anklickens werden die Formulardaten an die Datei *pruef.pl* gesendet.

9.2.11 Ereignisse bei Formularelementen

Ereignisse übergeben

Durch den Einsatz der **handleEvent()**-Methode kann ein Ereignis an ein Element übergeben werden, welches wiederum auf das übergebene Ereignis reagieren kann. Ausführlichere Informationen zum Thema Event-Handling finden Sie im Abschnitt 4.9 ab Seite 134. An dieser Stelle soll die Ereignisbehandlung bezüglich des **elements**-Objekts lediglich anhand des folgenden Beispiels veranschaulicht werden. Hierin befinden sich zwei einzeilige Eingabefelder sowie ein Formular-Button. Gibt der Anwender innerhalb des ersten Eingabefeldes eine Zeichenfolge ein, wird diese innerhalb des zweiten Feldes in den Tastaturcode umgewandelt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Netscape = window.Event ? true : false;
function Ereignisse(e)
{
  if (Netscape)
    var Tastatur = e.which
  else
    if (e.type == "keypress")
      var Tastatur = e.keyCode
    else
      var Tastatur = e.button;
  if (e.type == "keypress")
    document.Formular.Ausgabe.value = "keypress: Code=" + Tastatur + ",
Zeichen=" + String.fromCharCode(Tastatur)
  else
    document.Formular.Ausgabe.value = "click: Code=" + Tastatur;
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
Eingabe: <INPUT  type="text" size="20" onKeypress="Ereignisse(event)"><BR>
```

```
Ausgabe: <INPUT  type="text" name="Ausgabe" value="">
<INPUT  type="button" value="anklicken"
onclick="if (Netscape || window.event) Ereignisse(event)">
</FORM>
</BODY>
</HTML>
```

Listing 9.19: Der Keycode der gedrückten Taste wird in dem zweiten Eingabefeld angezeigt.

Nach Microsoft-Syntax lässt sich durch den Einsatz des JScript-Schlüsselworts **keyCode** der dezimale Wert einer gedrückten Taste ermitteln. Genau dieses wird im ersten Teil des Scripts getan. Im zweiten Schritt wird über die String-Methode **fromCharCode()** die eingegebene Zeichenkette als Latin-1-Zeichensatz interpretiert und der korrekte Zeichenwert wird wieder angezeigt. So ergibt sich beispielsweise aus der Eingabe des Zeichens *s* folgendes Szenario: Innerhalb des zweiten Eingabefeldes wird der Keycode `100` angezeigt und dieser anhand der **fromCharCode()**-Methode wiederum in das *s* umgewandelt.

Abbildung 9.4: Der Tastencode wird angezeigt.

9.2.12 Text in einem Eingabefeld selektieren

Anhand der **select()**-Methode kann der gesamte Text eines Eingabefeldes selektiert werden. Es wird kein Parameter erwartet. Die **select()**-Methode kann auf die folgenden Elemente angewandt werden:

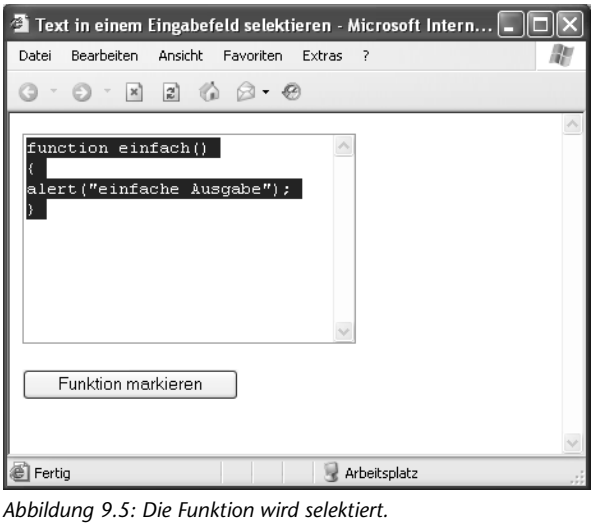
Einsatzfelder

- einzeilige Eingabefelder
- Passwort-Felder
- mehrzeilige Eingabefelder

Eingesetzt wird die **select()**-Methode häufig auf Seiten für Entwickler. Hier wird dann beispielsweise JavaScript-Code innerhalb eines mehrzeiligen Eingabefeldes angezeigt. Der Anwender kann diesen anschließend durch Anklicken eines Verweises selektieren und somit problemlos und schnell kopieren. Um diesem Beispiel Rechnung zu tragen, bedient sich

die folgende Syntax diesen Szenarios. Auch hier wird innerhalb eines mehrzeiligen Eingabefeldes eine JavaScript-Funktion angezeigt. Durch Anklicken des sich unter diesem Feld befindlichen Formular-Buttons wird der Script-Bereich selektiert. Ein anschließendes Kopieren des Feld-Inhalts kann somit unkompliziert vorgenommen werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function markiereInhalt()
{
document.Formular.Feld.select();
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<textarea rows="10" cols="30" name="Feld">
function einfach()
{
alert("einfache Ausgabe");
}
</textarea>
<BR>
<INPUT  type="button" onclick="markiereInhalt()" value="Funktion markieren">
</FORM>
</BODY>
</HTML>
```



Die Funktion wird markiert.

Abbildung 9.5: Die Funktion wird selektiert.

Durch Anklicken des Formular-Buttons wird die Funktion `markiereInhalt()` aufgerufen. Hierin wird zunächst über **`document.`**`Formular.Feld` auf das mehrzeilige Eingabefeld zugegriffen. Die Notation der **`select()`**-Methode bestimmt nun, dass der Feldinhalt selektiert werden soll.

9.3 Auswahllisten

Listen mit festgelegten Einträgen

Bei dem **`options`**-Element handelt es sich um ein Unterobjekt von **`elements`**. Durch **`options`** können Sie auf Auswahllisten zugreifen. Bei Auswahllisten handelt es sich um Listen mit festgelegten Einträgen. Hieraus kann der Anwender einen oder mehrere Einträge auswählen. Für den Zugriff auf Auswahllisten stehen wiederum verschiedene Syntaxformen zur Verfügung. Hierbei wird zwischen dem Zugriff auf die Auswahlliste und dem Zugriff auf die Einträge einer solchen Liste unterschieden.

```
document.forms[#].elements[#].options[#].Eigenschaft
document.forms[#].elements[#].Eigenschaft
```

Wollen Sie auf einen Eintrag der Auswahlliste zugreifen, müssen Sie bei der aufgeführten Syntaxform zunächst die Anweisung **`document.forms`** notieren. Innerhalb der eckigen Klammern wird durch eine Zahl angegeben, welches Formular angesprochen werden soll. Hieran schließt sich das **`elements`**-Objekt an. In der eckigen Klammer wird wieder eine Zahl erwartet. Durch diese wird bestimmt, das wievielte Element innerhalb des Formulars angesprochen werden soll. Über das **`options`**-Objekt wird auf die Auswahlliste zugegriffen. Der wievielte Eintrag dieser Liste angesprochen werden soll, wird über die Zahl innerhalb der eckigen Klammern festgelegt. Soll also beispielsweise der erste Eintrag angesprochen werden, muss hier der Wert *`0`* notiert werden. Abschließend wird die gewünschte Eigenschaft angegeben. Um auf die Auswahlliste, nicht aber auf einen bestimmten Eintrag zuzugreifen, wird das **`options`**-Objekt weggelassen. Die übrige Notation ist identisch. Die folgende Syntax soll das zuvor Beschriebene verdeutlichen.

```
document.forms[0].elements[0].options[2].value = "Taschenuhr"
document.forms[2].elements[1].selected = 3
```

Durch die erste Zeile wird auf das erste Element des ersten Formulars zugegriffen. Zusätzlich wird über die Anweisung **`options[`**`2`**`]`** bestimmt, dass der dritte Eintrag der Auswahlliste angesprochen wird. Als Eigenschaft wird hier **`value`** eingesetzt. Die zweite Zeile greift auf das zweite Element des dritten Formulars zu. Wir nehmen an, dass es sich hierbei um eine Auswahlliste handelt. Als Eigenschaft wird **`selected`** verwendet.

Auf Auswahllisten können Sie auch durch die Verwendung von Namen zugreifen. Hierbei müssen Sie die Namen des Formulars und des Ele-

ments angeben. Der hier zu notierende Name ergibt sich aus den **name**-Attributen des **<FORM>**- und des **<select>**-Tags. Auch hierbei stehen wieder zwei verschiedene Syntaxformen zur Verfügung. Deren jeweiliger Einsatz hängt davon ab, ob auf die Auswahlliste oder einen Eintrag dieser Liste zugegriffen werden soll.

```
document.Formularname.Elementname.options[#].Eigenschaft
document.Formularname.Elementname.Eigenschaft
```

In der ersten der beiden folgenden Zeilen wird auf die Auswahlliste `Autoren` in dem Formular `Formular` zugegriffen. Um den dritten Eintrag der Auswahlliste anzusprechen, wird die Anweisung **options[2]** verwendet. Als Eigenschaft wird hier **text** eingesetzt. Durch die zweite Zeile wird ausschließlich auf die Auswahlliste, nicht aber auf einen bestimmten Listeneintrag zugegriffen. Zu Demonstrationszwecken wird hier die **selectedIndex**-Eigenschaft verwendet.

```
document.Formular.Autoren.options[2].text = "Hornby";
document.Formular.Autoren.selectedIndex = 3;
```

Beachten Sie, dass die Einträge einer Auswahlliste nicht über deren Namen angesprochen werden können. Das liegt daran, dass Auswahllisten als Arrays behandelt werden. Es sind demnach ausschließlich Indexnummern zulässig.

9.3.1 Vorausgewählten Eintrag ermitteln

entscheidend ist das **selected**-*Attribut*

Die Eigenschaft **defaultSelected** speichert, ob ein Listeneintrag als vorausgewählt definiert wurde. In diesem Zusammenhang spielt die Verwendung des **selected**-Attributs innerhalb des **<OPTION>**-Tags die entscheidende Rolle. Dieses Attribut stellt einen Listeneintrag als vorausgewählt, also markiert, dar. Eingesetzt wird **selected** immer dann, wenn dem Anwender ein bestimmter Eintrag „schmackhaft“ gemacht werden soll. Die **defaultSelected**-Eigenschaft kennt die folgenden beiden Eigenschaften:

mögliche Werte

- **true** – Der Eintrag ist vorausgewählt.
- **false** – Der Eintrag ist nicht vorausgewählt.

Beachten Sie, dass trotz eines durch **selected** ausgezeichneten Listeneintrags durch den Anwender dennoch ein anderer ausgewählt werden kann. Die folgende Syntax lässt sich vortrefflich dazu verwenden, um den Anwender auf leichte Weise mehrere Listeneinträge gleichzeitig markieren zu lassen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
```

```
<!--
function markiereBuecher()
{
if(document.Formular.Buecher.options[0].defaultSelected == true)
     document.Formular.Buecher.options[1].selected=true;
 document.Formular.Buecher.options[2].selected=true;
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Buecher" size="4" multiple>
<OPTION selected>Die Kammer</OPTION>
<OPTION>Der Regenmacher</OPTION>
<OPTION>Die Akte</OPTION>
<OPTION>Letzte Instanz</OPTION>
</SELECT>
<BR>
<INPUT  type="button" onclick="markiereBuecher()" value="Grisham
B&uuml;cher">
</FORM>
</BODY>
</HTML>
```

Listing 9.20: Die Einträge 1-3 werden selektiert.

Das spezifizierte Formular beinhaltet eine Auswahlliste sowie einen Formular-Button. Durch das Anklicken dieses Buttons wird die Funktion `markiereBuecher()` aufgerufen. Diese prüft zunächst, ob der erste Listeneintrag vorausgewählt ist. Wird hier der Wert **`true`** zurückgeliefert, werden zusätzlich der zweite und dritte Eintrag selektiert.

Abbildung 9.6: Die Grisham-Bücher wurden selektiert.

9.3.2 Anzahl der Auswahlmöglichkeiten

size *bleibt ohne Bedeutung*

Wie viele Einträge innerhalb einer Auswahlliste tatsächlich definiert wurden, lässt sich über die **length**-Eigenschaft ermitteln. Beachten Sie, dass es sich hierbei nicht um die auf den ersten Blick dargestellte Anzahl handeln muss. Vielmehr werden auch solche Einträge berücksichtigt, die dem Anwender erst nach dem Anklicken der Auswahlliste angezeigt werden. Das HTML-Attribut **size** im einleitenden **<SELECT>**-Tag hat auf das Ergebnis der **length**-Eigenschaft demnach also keinerlei Auswirkungen.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Schriftsteller" size=1>
<OPTION>Goethe</option>
<OPTION>Schiller</option>
<OPTION>Hesse</option>
<OPTION>Balzac</option>
</SELECT>
</FORM>
<SCRIPT type="text/JavaScript">
document.write(+ document.Formular.Schriftsteller.length + "
Auswahlm&ouml;glichkeiten stehen bereit");
</SCRIPT>
</BODY>
</HTML>
```

Listing 9.21: Die Anzahl der Listeneinträge werden in das Dokument geschrieben.

Die aufgeführte Syntax beschreibt, wie sich die **length**-Eigenschaft für eine effizientere Nutzerführung nutzen lässt. Zwar erkennt der Anwender auf den ersten Blick nicht, wie viele Einträge innerhalb der Auswahlliste spezifiziert wurden, durch das eingefügte JavaScript wird dies dann aber doch ermöglicht. Und zwar unabhängig davon, dass die Größe dieser Auswahlliste auf 1 gesetzt wurde. Unterhalb des Formulars befindet sich ein Script-Bereich. Dieser wurde deshalb erst nach der Auswahlliste eingefügt, da diese Anweisung direkt nach dem Aufruf der Datei ausgeführt wird. Befände sich dieser Scriptbereich also vor der eigentlichen Auswahlliste, würde eine Fehlermeldung angezeigt werden. Über `Schriftsteller.length` wird die Anzahl der Einträge innerhalb der Auswahlliste ermittelt. Der hierbei ermittelte Wert wird über **write()** dynamisch in das Dokument geschrieben.

9.3.3 Wurde ein Listeneintrag selektiert?

Die **selected**-Eigenschaft speichert, ob und, wenn ja, welcher Eintrag innerhalb einer Auswahlliste selektiert ist. So lässt sich hierdurch beispielsweise überprüfen, ob der Anwender einen in jedem Fall zu selektierenden Listeneintrag ausgewählt hat. Ob ein Listeneintrag ausgewählt wurde, wird anhand der beiden boolschen Werte **true** und **false** überprüft. Deren jeweilige Bedeutung stellt sich folgendermaßen dar:

mögliche Werte

- **true** – Der Listeneintrag ist selektiert.
- **false** – Der Listeneintrag ist nicht selektiert.

Da die **selected**-Eigenschaft nicht nur das Lesen, sondern auch das Ändern von Werten gestattet, lassen sich hierdurch ganz gezielt Listeneinträge selektieren. Diesem Prinzip folgt die nachstehende Syntax. Dem Anwender wird eine Auswahlliste mit vier Einträgen, die alle sichtbar sind, angezeigt. In dieser Liste befinden sich vier Buchtitel. Zwei dieser Bücher wurden von John Grisham verfasst. Durch Anklicken eines Formular-Buttons werden exakt diese beiden Listeneinträge selektiert. Der Anwender erkennt also sofort, welche Bücher von John Grisham und welche von anderen Autoren verfasst wurden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function markiereBuecher()
{
document.Formular.Buecher.options[0].selected = true;
document.Formular.Buecher.options[2].selected = true;
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Buecher" size="4" multiple>
<OPTION>Die Kammer</option>
<OPTION>Lugal</option>
<OPTION>Die Akte</option>
<OPTION>Letzte Instanz</option>
</SELECT>
<BR>
<INPUT  type="button" onclick="markiereBuecher()" value="Grisham
B&uuml;cher">
</FORM>
```

```
</BODY>
</HTML>
```

Listing 9.22: Der erste und dritte Eintrag werden selektiert.

Die Funktion `markiereBuecher()` wird durch das Anklicken des Formular-Buttons ausgelöst. Hierin wird auf den ersten und den dritten Listeneintrag zugegriffen. Dies geschieht über die im Zusammenhang mit dem **`option`**-Objekt bekannte Adressierung. Durch **`selected = true`** wird festgelegt, dass die beiden Listeneinträge selektiert werden sollen.

9.3.4 Index der aktuellen Auswahl

Listeneinträge werden anderes angesprochen

Mittels der **`selectedIndex`**-Eigenschaft kann ermittelt werden, ob und, wenn ja, welcher Listeneintrag einer Auswahlliste selektiert wurde. Wurde kein Eintrag ausgewählt, wird der Wert **–1** gespeichert. Beachten Sie, dass es sich beim Ergebnis um das gleiche wie bei der Verwendung der **`selected`**-Eigenschaft handelt. Der einzige Unterschied zwischen beiden Eigenschaften liegt in der Art, wie die Listeneinträge angesprochen werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function selektiereEintrag()
{
 if(document.Formular.Auswahl.selectedIndex == 1)
  alert("So geht es aber nicht")
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Auswahl" size="3">
<OPTION>ja</OPTION>
<OPTION>weiss nicht</OPTION>
<OPTION>nein</OPTION>
</select>
<INPUT  type="button" onclick="selektiereEintrag()" value="absenden">
</FORM>
</BODY>
</HTML>
```

Listing 9.23: Der mittlere Eintrag darf nicht selektiert werden.

Durch Anklicken des Formular-Buttons wird die Funktion `selektiere-Eintrag()` ausgelöst. Diese überprüft, ob der zweite Eintrag der Auswahlliste `Auswahl` selektiert wurde. Ist dies der Fall, wird ein definierter Text durch die **`alert()`**-Methode ausgegeben.

9.3.5 Text einer Auswahlliste speichern

Listeneinträge dynamisch verändern

Welcher Inhalt einer Auswahlliste dem Anwender angezeigt wird, kann über die Eigenschaft **`text`** festgelegt werden. Prinzipiell ist diese Eigenschaft mit dem Inhalt des HTML-Tags **`<OPTION>`** zu vergleichen. Anhand der **`text`**-Eigenschaft kann also demnach der Text eines Listeneintrags geschrieben bzw. gelesen werden. Sinnvoll ist die **`text`**-Eigenschaft vor allem im Hinblick auf das dynamische Ändern von Listeneinträgen. In dem folgenden Beispiel wird gezeigt, wie sich der Inhalt eines einzeiligen Eingabefeldes dafür nutzen lässt, einen bestehenden Listeneintrag mit einem neuen Wert zu ersetzen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function neuerEintrag()
{
document.Formular.Inhalt.options[2].text =
document.Formular.Eingabe.value;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<select name="Inhalt" size="3">
<OPTION>Corporate Identity</option>
<OPTION>Kommunikation</option>
<OPTION>Was fehlt noch</option>
</select>
<INPUT  type="text" name="Eingabe" value="Public Relations">
<INPUT  type="button" onclick="neuerEintrag()" value="einf&uuml;gen">
</FORM>
</BODY>
</HTML>
```

Listing 9.24: Schreibend auf Auswahllisten zugreifen

Innerhalb der aufgeführten Syntax befindet sich ein Formular, in welchem wiederum eine Auswahlliste, ein einzeiliges Eingabefeld sowie ein

Button definiert sind. Durch das Anklicken des Buttons wird die Funktion `neuerEintrag()` ausgelöst. Darin wird zunächst über die Anweisung `Inhalt.options[2]` auf den dritten Eintrag der Auswahlliste `Inhalt` zugegriffen und wir dieser mit dem Wert des einzeiligen Eingabefeldes gleichgesetzt.

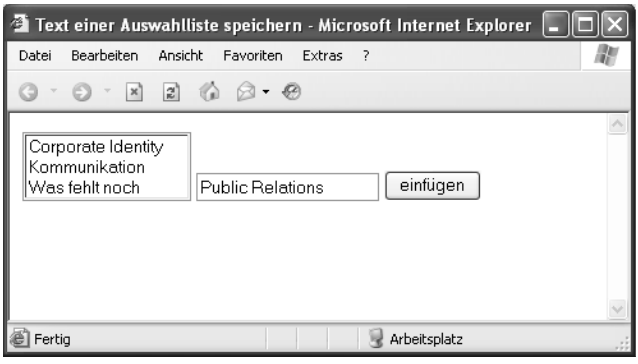


Abbildung 9.7: Die Liste nach dem Laden der Seite

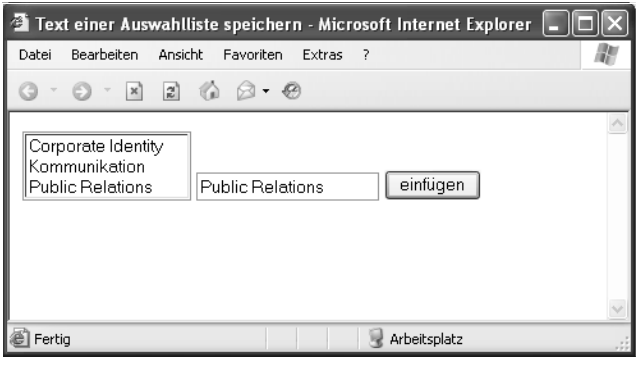


Abbildung 9.8: Der neue Eintrag wurde hinzugefügt.

9.3.6 Absendewert speichern

Nicht der gezeigte, sondern ein interner Wert wird versandt

Die **value**-Eigenschaft liest den internen Absendewert eines Listeneintrags aus. Hierbei bezieht sich diese Eigenschaft auf den innerhalb des **<OPTION>**-Tags notierten Wert des **value**-Attributs. Die Verwendung des **value**-Attributs des **<OPTION>**-Tags dient hierbei dazu, bestimmen zu können, dass nicht der dem Anwender angezeigte Text, sondern der vergebene Wert des **value**-Attributs versendet wird. Der Vorteil hiervon liegt auf der Hand. Denn gerade im Hinblick auf lange Listeneinträge ist die Weiterverarbeitung der Formulardaten nicht immer ganz einfach und übersichtlich. Um dieses Problem umgehen zu können, kann einem extrem langen Listeneintrag ein kurzer interner Absendewert zugewie-

sen werden. Die nachstehende Beispielsyntax beinhaltet eine Auswahlliste mit drei Einträgen. Jedem dieser Einträge wurde ein interner Absendewert zugewiesen. Selektiert der Anwender einen dieser Einträge, wird dessen interner Absendewert in einem Meldungsfenster ausgegeben. In diesem Beispiel hat dies zur Folge, dass dem Anwender die Autoren der innerhalb der Auswahlliste notierten Bücher angezeigt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeAutor()
{
for(var i=0; i<document.Formular.Liste.length;i++)
if(document.Formular.Liste.options[i].selected == true)
alert(document.Formular.Liste.options[i].value);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Liste" size="3" onchange="zeigeAutor()">
<OPTION value="Lindsay">Abgr&uuml;ndig</OPTION>
<OPTION value="Fielding">Lauf Jane, lauf</OPTION>
<OPTION value="Follet">Die Nadel</OPTION>
</SELECT>
</FORM>
</BODY>
</HTML>
```

Listing 9.25: Der Autor des jeweiligen Buches wird nach dem Anklicken eines Eintrags in einem Meldungsfenster ausgegeben.

Über den Event-Handler **onchange** wird die Funktion `zeigeAutor()` aufgerufen. Dieser überprüft anhand einer **for**-Schleife, welcher der Listeneinträge selektiert wurde. Anschließend wird auf den Listeneintrag über die Syntax **doucment.**`Formular.Liste`**.option[]** zugegriffen. Um den internen Absendewert zu ermitteln, wird die **value**-Eigenschaft verwendet. Anschließend wird der ermittelte Wert über die **alert()-Methode** ausgegeben.

9.4 Automatisierte E-Mail

Alle Angaben werden in einem Dialogfenster angezeigt.

Durch einfache HTML-Syntax können E-Mail-Verweise so konstruiert werden, dass neben dem Empfänger u.a. auch der Betreff und der Inhalt vordefiniert sind. Diese Möglichkeiten sind zwar bereits recht nützlich, sie lassen sich durch JavaScript aber noch besser nutzen. Das folgende Beispiel bietet dem Anwender beim Laden der Seite einen Formular-Button an. Dessen Anklicken öffnet nacheinander mehrere Dialogfenster. In jedem dieser Fenster wird ein bestimmter Wert, wie beispielsweise der Empfänger und der Betreff der Nachricht, abgefragt. Im letzten Schritt werden die notwendigen Daten zur Kontrolle nochmals angezeigt. Bestätigt der Anwender die Eingaben, wird das installierte E-Mail-Programm gestartet und die Nachricht kann versandt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function automatisierteEmail()
{
Email=prompt("E-Mail-Adresse: ","server@myserver.de");
Betreff=prompt("Betreffzeile: ","Hallo");
Inhalt=prompt("Nachricht: ","Eine Nachricht");
if (confirm
("Wollen Sie die E-Mail "+Email+" mit dem Betreff "+Betreff+" und dem Inhalt
"+Inhalt+" tatsächlich abschicken?")
==true)
  {
parent.location.href=
'mailto:'+Email+'?body='+Inhalt+'&amp;subject='+Betreff+'';
   }
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT  type=button value="E-Mail ausfüllen" onclick="automatisierteEmail()">
</FORM>
</BODY>
</HTML>
```

Listing 9.26: Die eingegebenen Werte können als E-Mail versandt werden.

Durch Anklicken des Formular-Buttons wird die `automatisierteEmail()`-Funktion ausgelöst. Mittels der Definition der drei Variablen `Email`, `Betreff` und `Inhalt` werden drei Dialogfenster nacheinander angezeigt. Für

die Generierung von Dialogfenstern wird die `prompt()`-Methode des `window`-Objekts verwendet. Als erster Parameter wird ein Aufforderungstext und als zweiter die Feldvorbelegung angegeben. Im letzten Schritt werden die eingegebenen Daten zur Kontrolle in einem Dialogfenster angezeigt. Der Anwender kann nun durch Anklicken des OK-Buttons die Eingaben bestätigen. Das führt zum Öffnen des installierten E-Mail-Programms. Wird der Abbrechen-Button angeklickt, geschieht nichts.

Abbildung 9.9: Die eingegebenen Werte werden angezeigt.

9.5 Formulareingaben überprüfen

Falsche Angaben können kostenintensiv sein.

Formulare dienen zumeist der Einholung von Nutzerdaten. So kann die E-Mail-Adresse des Anwenders für Sie beispielsweise dann von Bedeutung sein, wenn Sie einen Newsletter anbieten. Denn gerade dann, wenn diese Daten automatisch in eine Datenbank geschrieben werden, sind fehlerhafte Angaben ärgerlich. Mittels JavaScripts können Sie die Fehlerhäufigkeit von Eingaben mindern. Dies gilt freilich nur dann, wenn ein Anwender versehentlich falsche Daten eingibt. Beabsichtigt falsche Eingaben können Sie durch JavaScript nicht herausfinden. Dieser Abschnitt soll Ihnen zeigen, wie Sie falsche Werte in Eingabefeldern aufspüren können. Wobei der Fokus hier auf tatsächlich unbeabsichtigten Fehlern liegt. Es wird hier beispielsweise gezeigt, wie Sie überprüfen können, ob der Anwender ein Feld nicht ausgefüllt hat. Ob der eingetragene Text nun tatsächlich sinnvoll ist, können Sie indes nicht überprüfen.

9.5.1 E-Mail-Feld

Perfekte Lösungen gibt es nicht.

Auf vielen Internetseiten werden Dienste angeboten. Dabei kann es sich um Newsletter u.Ä. handeln. Das Einzige, was hierfür vom Anwender erwartet wird, ist, dass er seine E-Mail-Adresse angibt. Da diese häufig an potente Kunden weiterverkauft werden, scheuen sich viele Nutzer ihre wahre E-Mail-Adresse anzugeben. Wie problematisch dies ist, soll Ihnen folgender Fall beschreiben. Bei meinem Arbeitgeber, einem Softwareunternehmen, können sich Anwender kostenlos eine 30-Tage-Testversion einer Zeichensoftware downloaden. Als einzige Gegenleistung wird die Angabe einer E-Mail-Adresse erbeten. Die Auswertung der Datenbank ist jedes Mal eine wahre Freude. Von circa 1000 Anwendern, von denen die Software heruntergeladen wird, stimmen etwa 70-80 Adressen. Der Rest

sind schlicht und ergreifend Phantasie-Adressen. Die Arbeit des Ausfilterns falscher Einträge nimmt demzufolge eine enorme Zeit in Anspruch. (Aus diesem Grund danke ich an dieser Stelle allen Praktikanten, die diese Aufgabe mit dem notwendigen Ehrgeiz angehen.) Aber genug abgeschweift. Das beschriebene Problem macht deutlich, dass eine Lösung gefunden werden muss. Natürlich ist es so, dass Sie nicht alle falschen Einträge vor dem Absenden eines Formulars herausfiltern können. Wir wollen uns in dem folgenden Beispiel vielmehr darum kümmern, grob fahrlässige Fehler zu finden. Ein Eintrag wie beispielsweise *gehtdichnichtsan* wird von diesem Programm nicht akzeptiert. Wir überprüfen die vom Anwender eingegebene E-Mail-Adresse auf mehrere Kriterien. Ist das Feld leer? Wurde ein @ eingegeben? Enthält die Adresse einen Punkt? Natürlich ist auch diese Variante noch nicht perfekt. Sie hilft aber dennoch, die gröbsten Fehleinträge zu finden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function checkEmail() {
 vali = true;
  if (!checkEmail2(document.form.email.value))
{
validity = false; alert('Das ist keine gültige E-Mail-Adresse!!');
}
  return validity;
}
function checkEmail2(mail) {
  if ((mail == "")
    || (mail.indexOf ('@') == -1)
    || (mail.indexOf ('.') == -1))
      return false;
  alert("korrekt");
  return true;
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="form" onsubmit="return checkEmail()">
<INPUT  type="text" size=20 name="email">
<INPUT  type="submit" name="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.27: Handelt es sich bei dem eingegebenen Wert um eine gültige E-Mail-Adresse?

Die Funktion `checkEmail()` wird durch den Event-Handler **`onsubmit`** ausgelöst. Dieser überprüft, ob die Bedingungen der Funktion `checkEmail2()` erfüllt sind. Ist dies nicht der Fall, wird eine Fehlermeldung ausgegeben. Angewandt wird diese Funktion auf das Eingabefeld `email`. Dieses Feld wird der `checkEmail2()`-Funktion als Parameter `mail` übergeben. Innerhalb der Funktion wird über `mail == ""` überprüft, ob überhaupt ein Eintrag vorliegt. Um zu überprüfen, ob die beiden Zeichen `@` und `.` vorkommen, wird die **`indexOf()`**-Methode verwendet. Diese liefert, wenn die Suche nach den angegebenen Zeichen erfolglos war, den Wert **–1** zurück. Tritt dieser Rückgabewert auf, wird die Fehlermeldung ausgegeben. Verlief die Suche nach beiden Zeichen erfolgreich, wird das Wort `korrekt` mittels der **`alert()`**-Methode angezeigt.

Abbildung 9.10: Eine ungültige E-Mail-Adresse

9.5.2 URL überprüfen

Vor mutwillig falschen Angaben gibt es keinen Schutz.

Die hier vorgestellte Syntax ähnelt der E-Mail-Funktion aus dem letzten Abschnitt. Nur wird hier nicht die Gültigkeit einer E-Mail-Adresse, sondern die eines URLs überprüft. Es gilt auch hier die gleiche Motivation, nämlich die, dass es dem Anwender erschwert werden soll, ungültige Daten einzugeben. Aber ebenso wie bei der E-Mail-Funktion sind Sie auch hierbei nicht vor mutwillig falsch eingetragenen Werten gefeit. Zunächst müssen wir uns einen geeigneten Logarithmus überlegen, mit dem ein gültiger von einem ungültigem URL unterschieden werden kann. Wir wissen, dass das URL-Feld nicht leer sein darf. Zudem sollte jeder URL mit der Zeichenkette *`http://`* beginnen und einen Punkt enthalten. Diese drei Kriterien sollen für unsere Zwecke ausreichend sein.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function urlPruefer()
{
test = true;
if (!urlPruefer2(document.form.url.value))
        { test = false; alert('Kein gültiger URL!'); }
}
function urlPruefer2(eingabe)
{
  if ((eingabe == "")
    || (eingabe.indexOf ('http://') == -1)
    || (eingabe.indexOf ('.') == -1))
      return false;
  alert("korrekt");
  return
true;
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="form" onsubmit="return urlPruefer()">
<INPUT  type="text" size=20 name="url">
<INPUT  type="submit" name="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.28: Handelt es sich bei dem eingegebenen Wert um einen gültigen URL?

Die Funktion `urlPruefer()` wird durch den Event-Handler **`onsubmit`** ausgelöst. In diesem wird überprüft, ob die Bedingungen der Funktion `urlPruefer2()` erfüllt sind. Als Parameter wird der `urlPruefer2()`-Funktion der Wert des Feldes `url` übergeben. Innerhalb der Funktion wird zunächst über die Anweisung `eingabe` **`== ""`** überprüft, ob das Listenfeld leer ist. Um herauszufinden, ob die beiden notwendigen Elemente `http://` und der Punkt vorkommen, wird die **`indexOf()`**-Methode eingesetzt. Diese liefert immer dann den Rückgabewert **–1**, wenn der angegebene Suchbegriff nicht gefunden wurde. Werden diese Bedingungen erfüllt, wird die in der Funktion `urlPruefer()` angegebene Fehlermeldung ausgegeben. Anderenfalls erfolgt die Ausgabe des Wortes `korrekt` mittels der **`alert()`**-Methode.

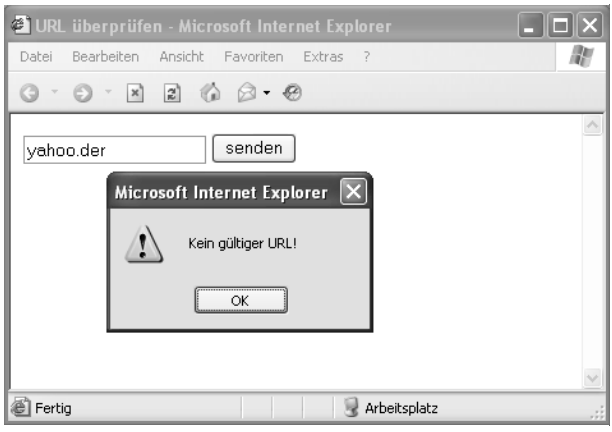


Abbildung 9.11: Fehlermeldung, da kein gültiger URL

9.5.3 Nur Buchstaben

Offensichtlich falsche Namen können ausgesiebt werden.

In vielen Formularen wird verlangt, dass der Anwender seinen Namen angibt. Wer bereits einmal Datenbanken, in denen die Namen der Anwender eingetragen wurden, ausgewertet hat, kennt das Problem: Viele Namen wurden nicht korrekt geschrieben. Nun ist es natürlich so, dass Sie keine Möglichkeit besitzen, einen Scherznamen von einem echten zu unterscheiden. Das folgende Script zeigt aber dennoch einen Weg, wie sich zumindest die gröbsten Fehler verhindern lassen. Hierzu wird überprüft, ob in dem Namen ein anderes Zeichen außer den Buchstaben von A bis Z und dem Bindestrich vorkommen. Ist dies der Fall, kann das Formular nicht versendet werden. Stattdessen wird eine Fehlermeldung ausgegeben. Nun ist es aber freilich so, dass Sie falsche Namen dennoch nicht von korrekten unterscheiden können. Schließlich muss ein eingetragener Name, der nur aus Buchstaben besteht und somit syntaktisch korrekt ist, nicht der richtige Name des Anwenders sein. Immerhin hilft das nachfolgende Programm dabei, solche Namen auszufiltern, die offensichtlich falsch, da mit Ziffern und Sonderzeichen ausgestattet, sind.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function pruefeName()
{
var zulaessig="abcdefghijklmnopqrstuvwxyzäöüß-";
istesText=true;
Textkontrolle=document.Formular.elements[0].value.toLowerCase();
if(Textkontrolle.length==0)
istesText=false;
```

```
else
for(n=0;n<Textkontrolle.length;n++)
{
if(zulaessig.indexOf(Textkontrolle.charAt(n))==-1)
istesText=false;
}
if(istesText)
 document.Formular.submit();
else
{
alert("Geben Sie bitte Ihren Namen an!" );
document.Formular.elements[0].focus();
}
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="mailto:my@provider.de">
   <INPUT  type=text name="Name">
   <INPUT  type="button" value="senden" onclick="pruefeName();">
</FORM>
</BODY>
</HTML>
```

Listing 9.29: Wurden tatsächlich nur Buchstaben eingegeben?

Innerhalb der Variablen `zulaessig` sind alle zulässigen Zeichen angegeben. Um nicht Groß- und Kleinbuchstaben angeben zu müssen, verwenden wir die **`toLowerCase()`**-Methode. Diese der Textkontrolle-Variablen zugewiesene Methode wandelt alle vorkommenden Buchstaben in Kleinbuchstaben um. Im ersten Schritt wird über die **`if`**-Abfrage überprüft, ob das Eingabefeld leer ist. Hierfür wird die Anweisung **`length==0`** verwendet. Ist diese Bedingung erfüllt, wird in einem Meldungsfenster eine Fehlermeldung ausgegeben. Ist das Feld nicht leer, wird die **`for`**-Schleife ausgeführt. Diese greift nacheinander auf die einzelnen Zeichen des Eingabefeldes zu und vergleicht diese mit den in der Variablen `zulaessig` spezifizierten zulässigen Zeichen. Die **`indexOf()`**-Methode wird hierbei dazu verwendet, um zu ermitteln, ob das vorhandene Zeichen mit einem der in der `zulaessig`-Variablen gespeicherten Zeichen übereinstimmt. Ist dies nicht der Fall, liefert diese Methode den Wert **–1** zurück. Wird dieser Rückgabewert für ein Zeichen geliefert, wird über die **`alert()`**-Methode eine Fehlermeldung ausgegeben. Wurden ausschließlich zulässige Zeichen eingegeben und ist das Eingabefeld nicht leer, wird das Formular mittels der **`submit()`**-Methode versendet.

9.5.4 Leeres Feld

Das Eingabefeld wird fokussiert.

Die Validierung von Nutzereingaben bezüglich von leeren Feldern ist eine der am weitesten verbreiteten Anwendungen für JavaScript im Zusammenhang mit Formularen. Handelt es sich doch hierbei um eine der Validierungsformen, die sich nahezu auf jedes Formular und jedes Eingabefeld anwenden lassen. Eingesetzt wird die in diesem Abschnitt vorgestellte Syntaxform häufig im Kontext mit Formularen bzw. Formularelementen, die zwar keine speziellen Zeichen, wohl aber das Notieren mindestens eines Zeichens verlangen. Eine Anwendung, die zum einen den Part des Überprüfens und darüber hinaus zusätzliche Raffinessen bereithält, beschreibt das folgende Beispiel.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function leeresFeld() {
var Name = document.Formular.Name.value;
if ((Name.length<1)) {
alert("Füllen Sie bitte das Feld aus.");
document
Formular.Name.value="Genau dieses hier!";
document.Formular.Name.focus();
return false;
}
else { return true; }
}
// -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" onsubmit="return leeresFeld()">
<INPUT  type="text" name="Name" size=40><BR>
<INPUT  type="submit" name="button" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.30: Ist das Eingabefeld leer?

Innerhalb des Formulars `Formular` wurden ein einzeiliges Eingabefeld sowie ein **`Submit`**-Button definiert. Innerhalb des Eingabefeldes soll der Anwender zumindest ein Zeichen notieren. Bei Missachtung dieses Wunsches soll ein **`alert`**-Fenster mit einem entsprechenden Hinweis ausgegeben werden. Als zusätzliche Motivation wird im Anschluss hieran das Eingabefeld mit einem festgelegten Text vorbelegt. Die hier-

für verantwortlich zeichnende Funktion lautet `leeresFeld()` und wird durch den Event-Handler **`onsubmit`** ausgelöst. Zunächst wird über die Anweisung `Name`**`.length<`**`1` überprüft, ob weniger als ein, also kein Zeichen, eingegeben wurde. Ist diese Bedingung erfüllt, wird ein **`alert()`**-Fenster mit dem Hinweis `Füllen Sie bitte das Feld aus.` ausgegeben. Nach dem Bestätigen dieses Fensters durch den OK-Button wird das Eingabefeld mit dem Namen `Name` über die Anweisung **`document.`**`Formular`**`.`**`Name`**`.value` angesprochen und mit einem Wert vorbelegt. Zusätzlich hierzu wird das Eingabefeld über die Anweisung **`focus()`** fokussiert. Für den Fall, dass der Anwender zumindest ein Zeichen notiert hat, wird das Formular ohne Beanstandung abgesendet. Hierbei spielt es keine Rolle, welches Zeichen eingegeben wurde.

9.5.5 Mehrere leere Textfelder

Alle Eingabefelder werden mittels einer **`for`***-Schleife überprüft.*

Im letzten Abschnitt haben Sie eine Möglichkeit zur Kontrolle eines einzeiligen Eingabefeldes kennen gelernt. Dieses Programm ist zwar lauffähig, bei der gewünschten Kontrolle mehrerer Eingabefelder funktioniert es aber nicht. In einem solchen Fall könnte man die Funktion nun mehrmals unter anderen Namen verwenden und jeweils ein anderes Feld überprüfen. Besser wäre hier allerdings die Verwendung einer **`for`**-Schleife. Nehmen wir Folgendes an: In einer HTML-Seite befinden sich drei einzeilige Formularfelder. Diese sollen vom Anwender in jedem Fall ausgefüllt werden. Sind beim Anklicken des **`Submit`**-Buttons nicht alle Felder ausgefüllt, soll eine Fehlermeldung ausgegeben werden. Dieses Szenario zeigt Abbildung 9.12.

Abbildung 9.12: Fehlermeldung, da das dritte Feld leer ist

Zunächst muss überlegt werden, in welcher Form alle Eingabefelder abgefragt werden sollen. Da die Anzahl der Felder bekannt ist, entscheiden wir uns für die Verwendung einer **for**-Schleife. Zusätzlich sollte das Programm sicherstellen, dass das Absenden des Formulars nur dann vorgenommen wird, wenn tatsächlich alle Felder ausgefüllt sind.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function alleFelder()
{
ausgefuellt=true;
for(i=0;i<3;i++)
{
if(window.document.Formular.elements[i].value == "")
ausgefuellt=false;
}
if(!ausgefuellt)
alert("Füllen Sie bitte alle Felder aus!"); return ausgefuellt;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
   <INPUT  type=text name="Name">
   <INPUT  type=text name="Email">
   <INPUT  type=text name="Telefon">
   <INPUT  type="submit" onclick="return alleFelder()">
</FORM>
</BODY>
</HTML>
```

Listing 9.31: Wurden tatsächlich alle Eingabefelder ausgefüllt?

Die Funktion `alleFelder()` wird durch Anklicken des **Submit**-Buttons ausgelöst. Der **for**-Schleife wird die Laufvariable `i` mit dem initialisierten Wert `0` zugewiesen. Als Abbruchkriterium wird **i<3** genutzt. Im nächsten Schritt werden über **elemente[i]** alle Eingabefelder nacheinander angesprochen. Durch die Anweisung **value == ""** wird überprüft, ob eines oder mehrere dieser Felder leer sind. Ist dies der Fall, wird eine Fehlermeldung ausgegeben.

9.5.6 Vorgegebene Werte überprüfen

eine Schulnote zwischen 1 und 6

Ähnliche Fälle werden Ihnen häufiger begegnen. Stellen Sie sich vor, dass Sie ein Formular mit verschiedenen Eingabefeldern entwickeln. In einem dieser Felder soll der Anwender eine Schulnote eingeben. Diese kann beispielsweise für die Bewertung Ihrer Seiten genutzt werden. Im folgenden Beispiel soll der eingegebene Wert daraufhin überprüft werden, ob es sich bei diesem tatsächlich um eine gültige Schulnote handelt. Führen wir uns also zunächst das Problem vor Augen. Damit eine Schulnote gültig ist, muss ein Wert zwischen 1 und 6 angegeben werden. Beachten Sie, dass in diesem Beispiel keine Überprüfung hinsichtlich Ganzzahlen vorgenommen wird. Auf Grund der Syntax ist also auch der Wert 1.5 eine gültige Schulnote. Aber wie gesagt, handelt es sich hierbei um ein Beispiel, welches das Prüfen von vorgegebenen Werten veranschaulichen soll. Mit dem in diesem Buch vermittelten Wissen können Sie weitere Kriterien für eine Abfrage schnell hinzufügen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Schulnote()
{
var Kontrolle = document.Formular
var Note = Kontrolle.Note.value
if (Note>=1 && Note<=6)
{
 return true
 }
else
 {
 alert("Diese Schulnote kann nicht stimmen!")
 return false
 }
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="mailto:myserver.myhost.com" onsubmit="return
Schulnote()">
Geben Sie bitte eine Schulnote von 1 bis 6 an:
<INPUT  type="text" name="Note">
<INPUT  type="submit" value="absenden">
</FORM>
</BODY>
</HTML>
```

Listing 9.32: Handelt es sich bei dem eingegebenen Wert um eine Zahl zwischen 1 und 6?

Die Funktion `Schulnote()` wird durch Anklicken des **`Submit`**-Buttons ausgelöst. Die Variable `Kontrolle` dient lediglich der Bequemlichkeit. Im gleichen Kontext ist die Deklaration der Variablen `Note` zu verstehen. Beide Variablen ermöglichen den Zugriff auf den Wert des Eingabefeldes `Note`. Das Hauptstück des Programms stellt die **`if`**-Abfrage dar. Diese überprüft, ob der eingegebene Wert **`>=`**`1` und **`<=`** `6` ist. Ist diese Bedingung erfüllt, kann das Formular versendet werden. Wurde kein dieser Abfrage entsprechender Wert eingegeben, gibt das Programm eine Fehlermeldung aus.

9.5.7 Mindestanzahl an Zeichen

Länderkürzel als Beispiel

Häufig kann es vorkommen, dass Eingaben in Formularfeldern auf eine bestimmte Mindestanzahl von eingegebenen Werten hin überprüft werden sollen. Dies kann beispielsweise bei dem Geburtsdatum, aber auch bei der Notation von Länderkürzeln der Fall sein. Anhand der nachfolgenden Syntax wird überprüft, ob innerhalb eines einzeiligen Eingabefeldes mindestens zwei Zeichen eingegeben wurden. Ist dies der Fall, kann das Formular an eine vorgegebene E-Mail-Adresse versandt werden. Bei Nichterfüllung der Bedingung wird ein Meldungsfenster geöffnet und das Formular wird nicht gesendet.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function ZeichenAnzahl()
{
Feld = document.forms[0].Eingabe.value
if (Feld.length<2)
 {
 alert("Das ist leider kein Länderkürzel!")
 return false
 }
else
 {
 return true
 }
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM action="mailto:name@provider.de" onsubmit="return ZeichenAnzahl()">
Geben Sie bitte das Länderkürzel an:
```

```
<INPUT  type="text" name="Eingabe">
<INPUT  type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.33: Wurden weniger als zwei Zeichen eingegeben?

Die Funktion `ZeichenAnzahl()` wird durch den Event-Handler **`onsubmit`** ausgelöst. Zunächst wird in der Funktion die Variable `Feld` deklariert. Als Wert wird dieser der Wert des einzeiligen Eingabefeldes zugewiesen. Die Überprüfung auf einen validen Wert wird über eine **`if`**-Abfrage realisiert. Hierzu wird als Bedingung die Angabe `Feld.length < 2` verwendet, wobei `Feld` den Wert des Eingabefeldes darstellt und die **`length`**-Eigenschaft die Anzahl der eingegebenen Zeichen ausliest. Ist diese Anzahl geringer als `2`, wird ein Meldungsfenster geöffnet. Ist der Wert gleich oder höher `2`, wird der **`else`**-Zweig der **`if`**-Abfrage angewandt und das Formular versendet.

9.6 Wurde ein Radio-Button selektiert?

anders als bei Eingabefeldern

Nicht nur der Zugriff auf Radio-Buttons wird in JavaScript anders als bei einzeiligen Eingabefeldern realisiert. Auch die Überprüfung, ob ein Radio-Button angeklickt wurde, stellt sich anders dar. Stellen wir uns folgendes Szenario vor: Sie entwickeln ein Formular, in dem sich mehrere Radio-Buttons befinden. Der Anwender soll in jedem Fall einen von diesen selektieren. Wird dies nicht getan, soll das Formular nicht versandt werden. Um auf alle Radio-Buttons zugreifen zu können, empfiehlt sich die Verwendung einer **`for`**-Schleife. Die Eigenschaft, mit welcher der Zustand von Radio-Buttons überprüft werden kann, ist **`checked`**. Im folgenden Beispiel befinden sich drei Radio-Buttons, von denen einer selektiert werden muss. Erst dann kann das Formular abgeschickt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function RadioKontrolle()
{
Namen=false;
for (i=0;i<3;i++) {
if (Formular.Autor[i].checked) Namen=true;
}
if (!Namen) {
alert('Wählen Sie bitte einen Autor!');
event.returnValue=false;
}
```

```
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="http://www.atmeins.de/cgi-bin/string"
method="post" onsubmit="RadioKontrolle();">
<INPUT  type="radio" name="Autor" value="ellis">Ellis
<INPUT  type="radio" name="Autor" value="roth">Roth
<INPUT  type="radio" name="Autor" value="hornby">Hornby
<INPUT  type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.34: Hat der Anwender einen Radio-Button selektiert?

Die `RadioKontrolle()`-Funktion wird durch den Event-Handler **`onsubmit`** ausgelöst. Innerhalb dieser Funktion werden alle drei Radio-Buttons mittels einer **`for`**-Schleife angesprochen. Als Abbruchkriterium der Schleife wird `i>3` verwendet. Da es sich um eine Gruppe von Radio-Buttons handelt, wird allen drei über das **`name`**-Attribut der gleiche Bezeichner, nämlich `Autor`, zugewiesen. Mittels der Anweisung `Autor[i]` kann nun nacheinander auf alle drei Radio-Buttons zugegriffen werden. Um zu überprüfen, ob ein Radio-Button selektiert wurde, wird die **`checked`**-Eigenschaft verwendet. Liefert diese den Wert **`true`**, wurde ein Radio-Button gewählt und das Formular kann versandt werden. Anderenfalls wird eine Fehlermeldung ausgegeben und das Formular wird nicht verschickt.

9.7 Validierung plus Fokussierung

erleichterte Fehlerkorrektur

Umfangreiche Formulare sind für den Anwender immer etwas schwierig zu handhaben. Dies gilt vor allem dann, wenn versehentlich ein Feld nicht ausgefüllt wurde, Sie aber festgelegt haben, dass das Formular nur versandt werden darf, wenn alle Felder ausgefüllt wurden. Befinden sich in dem Formular lediglich zwei Eingabefelder, ist das nicht ausgefüllte Feld schnell gefunden. Problematisch wird dies allerdings bei zwanzig oder dreißig Eingabefeldern. In einem solchen Fall müsste der Anwender die gesamte Seite durchsuchen. Besser ist es, ein nicht ausgefülltes Feld automatisch zu erkennen und zu fokussieren. Hierdurch wird der Cursor automatisch in das leere Eingabefeld gesetzt. Der Anwender kann das Feld somit ohne langwierige Suche ausfüllen. Einen solchen Fall beschreibt die nachstehende Syntax. Hierin werden drei Eingabefelder definiert. Das Programm ist so ausgerichtet, dass alle drei Felder da-

raufhin überprüft werden, ob sie leer sind. Ist das der Fall, wird eine Meldung mit einem jeweils anderen Hinweistext ausgegeben und das betreffende Feld fokussiert. Erst wenn in jedem der drei Felder ein Wert eingetragen wurde, kann das Formular verschickt werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
    <!--
    function Validierung()
    {
     var Formular = document.Kontrolle
        if (Formular.Name.value == "")
        {
            alert("Geben Sie bitte Ihren Namen an.");
            Formular.Name.focus();
            return false;
        }
        else if (Formular.Vorname.value == "")
        {
            alert("Geben Sie bitte Ihren Vornamen an.");
            Formular.Vorname.focus();
            return false;
        }
        else if (Formular.Email.value == "")
        {
            alert("Geben Sie bitte Ihre E-Mail-Adresse an.");
            Formular.Email.focus();
            return false;
        }
    }
    //  -->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Kontrolle" method="post" action="../../cgi-bin/email.pl?id"
onsubmit="return Validierung()">
Name : <INPUT  type="text" name="Name" size=15><BR>
Vorname : <INPUT  type="text" name="Vorname" size=15><BR>
E-Mail : <INPUT  type="text" name="Email" size=15><BR>
<INPUT  type="submit" value="senden">
</FORM>
</BODY>
</HTML>
```

Listing 9.35: Die nicht ausgefüllten Felder werden fokussiert.

Ausgelöst wird die Funktion `Validierung()` durch den Event-Handler **`onsubmit`**. Die Variable `Formular` ist lediglich eine Hilfsvariable, durch die der Schreibaufwand innerhalb des Programms gemindert wird. Sie dient dem Zugriff auf das Formular. Mittels einer **`if`**-Abfrage wird nun nacheinander überprüft, ob die Felder leer sind. Hierzu wird bei allen drei Feldern die Anweisung **`value == ""`** verwendet. Ist ein Feld leer, wird zunächst über **`alert()`** ein entsprechender Text ausgegeben. Um das betreffende Feld zu fokussieren, wird die **`focus()`**-Methode eingesetzt. Dieses Vorgehen wird auf alle drei Eingabefelder angewandt. Erst nachdem alle Felder einen Wert erhalten haben, wird das Formular an das Perl-Script gesandt.

9.8 Bestätigung in Popup-Fenster

Die Angaben können nochmals überprüft werden.

Worum es geht: Innerhalb eines Formulars befinden sich mehrere Eingabefelder. Diese sollen vom Anwender ausgefüllt werden. Eine Überprüfung, ob die Felder tatsächlich Werte enthalten, entfällt. Nach dem Anklicken des Formular-Buttons wird ein Popup-Fenster geöffnet. Hierin werden die eingetragenen Werte sowie ein Senden-Button angezeigt. Der Anwender hat somit nochmals die Möglichkeit, seine Eintragungen zu überprüfen. Ist er mit diesen zufrieden und klickt er den Button an, wird das Formular versendet. Das Beschriebene soll die Abbildung 9.13 visualisieren.

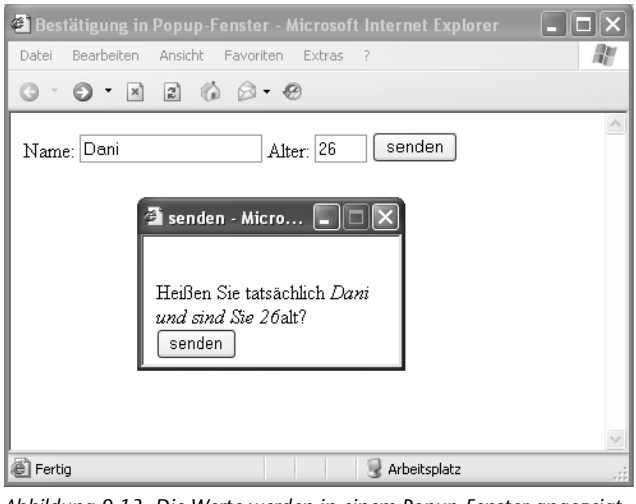


Abbildung 9.13: Die Werte werden in einem Popup-Fenster angezeigt.

Wie Sie sehen, wurden die Werte des Eingabefeldes tatsächlich übernommen. Die Lösung einer solchen Aufgabe erfordert einiges an Überlegung. Zunächst muss das Popup-Fenster realisiert werden. Hierzu bedienen wir uns der **`open()`**-Methode. Da dieses Fenster keinen statischen

Inhalt besitzt, muss der Fensterinhalt dynamisch erzeugt werden. Im nächsten Schritt müssen wir uns überlegen, wie wir das Formular des Hauptfensters aus dem Popup-Fenster heraus absenden können. All diese Fragen löst die folgende Syntax:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Bestaetigung() {
var Name = document.Formular.Name.value;
var Alter = document.Formular.Alter.value;
Anzeige = ("<HTML><HEAD></HEAD>");
Anzeige = (Anzeige + "<SCRIPT type='text/JavaScript'>function Sender()
{opener.Formular.submit();}</script>");
Anzeige = (Anzeige + "<BODY onblur='setTime-
out(\"self.focus()\",350)'><P><BR>");
Anzeige = (Anzeige + "<p>Heißen Sie tatsächlich <i>" + Name + " und sind Sie "
+ Alter + "</i>alt?");
Anzeige = (Anzeige + "<input type='button' onclick='Sender()'
value='senden'");
Anzeige = (Anzeige + "</BODY></HTML>");
Fenster=window.open("","NeuesFenster","width=200,height=100");
Fenster.document.writeln(Anzeige);
Fenster.document.close();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular" action="empfang.cfm">
<P>Name: <INPUT  type="text" size="20" name="Name"> Alter: <INPUT  type="text"
size="3" name="Alter">
<INPUT   type="button" onclick="Bestaetigung()" value="senden"> </P>
</FORM>
</BODY>
</HTML>
```

Listing 9.36: Die Deklaration einer Funktion in einem neuen Fenster

Um das Popup-Fenster zu öffnen, wird die `Bestaetigung()`-Funktion verwendet. Diese wird durch den Event-Handler ausgelöst. Innerhalb dieser Funktion werden zunächst die Werte der beiden Eingabefelder in den Variablen `Name` und `Alter` gespeichert. Der Inhalt des Popup-Fensters wird durch die Variable `Anzeige` definiert. Diese Variablendeklaration enthält neben dem HTML-Grundgerüst und herkömmlichem HTML-Code auch die Funktion `Sender()`. Diese greift über **`opener.`**`Formular` auf

das Formular des Hauptfensters zu. Damit dieses versendet werden kann, wird die **submit()**-Methode angewandt. Ausgelöst wird diese Funktion durch Anklicken des Formular-Buttons innerhalb des Popup-Fensters. Die Variable `Fenster` definiert das Aussehen des Popup-Fensters. Dessen Inhalt wird dynamisch über die Anweisung **writeln(**`Anzeige`**)** festgelegt. Die abschließend angegebene **close()**-Methode gibt an, dass der Dokumentinhalt geschlossen wird.

9.9 Fragen und Übungen

1. Schreiben Sie ein JavaScript-Programm, welches ein Formular nach 30 Sekunden automatisch abschickt. Tipp: Verwenden Sie die **setTimeout()**-Methode des **window**-Objekts und setzen Sie hier als zweiten Parameter den Wert `30000` für 30 Sekunden.

10 Grafiken

lernen

Ziele dieses Kapitels

Grafiken werden in HTML-Dateien über das **<IMG>**-Tag eingebunden. Als gängige Grafikformate haben sich mittlerweile GIF und JPEG herauskristallisiert. Bei der Verwendung von Grafiken ist darauf zu achten, dass diese nicht zu ladeintensiv sind. Als weitere Kriterien sollten das stete Vorhandensein von alternativen Texten durch das **alt**-Attribut sowie die Angaben zu Höhe und Breite der Grafik berücksichtigt werden. Durch JavaScript können Sie Grafiken dynamisch austauschen und wechselseitig anzeigen. Einige der zahlreichen Anwendungsmöglichkeiten für JavaScript bezüglich Grafiken werden in diesem Kapitel vorgestellt.

10.1 Zugriff auf Grafiken

Neue Grafik-Objekte werden automatisch erzeugt.

Grafiken werden aus JavaScript heraus über das **images**-Objekt angesprochen. Ein neues Grafik-Objekt wird automatisch erzeugt, wenn der WWW-Browser auf eine Grafik trifft. Es existieren zwei unterschiedliche Syntaxformen, um auf Grafiken zuzugreifen. Sie können hierbei zwischen der Verwendung von Indexnummern und Indexnamen wählen. Die folgende allgemein gültige Syntax zeigt, wie mittels Indexnummer auf eine Grafik zugegriffen und deren Eigenschaften und Methoden angesprochen werden können.

```
document.images[#].Eigenschaft
document.images[#].Methode()
```

Hinter der Anweisung **document.images** wird in der eckigen Klammer eine Zahl notiert. Durch diese wird angegeben, die wievielte Grafik innerhalb der Datei angesprochen werden soll. Durch den Wert *2* sprechen Sie die dritte Grafik an. Bei dieser Zählung wird jedes **<IMG>**-Tag, das in der Datei vorkommt, mitgezählt. Zwei Beispiele:

```
document.images[2].border
document.images[3].handleEvent()
```

Sie können auch per Indexnamen auf Grafiken zugreifen. Hierzu wird hinter dem **document**-Objekt der Grafikname notiert. Dabei muss der gleiche Name, der dem **name**-Attribut des **<IMG>**-Tags zugewiesen wurde, angegeben werden. Hieran schließt sich die gewünschte Eigenschaft oder Methode an.

```
document.Bildname.Eigenschaft
document.Bildname.Methode()
```

Durch die beiden folgenden Anweisungen wird auf die Grafik `Grafik` innerhalb des aktuellen Dokuments zugegriffen. Als Eigenschaft werden **width** und als Methode, übrigens die beim **images**-Objekt einzig verfügbare, **handleEvent()** angewandt.

```
document.Grafik.width
document.Grafik.handleEvent()
```

Die bislang gezeigten Syntaxformen helfen nur dann, wenn bereits eine Grafik vorhanden ist. Im Umgang mit Grafiken wird aber JavaScript erst dann richtig interessant, wenn neue Grafik-Objekte erzeugt werden können. Die folgende allgemein gültige Syntax wird für die Erstellung eines neuen Grafik-Objekts verwendet.

```
Grafikname = new Image()
```

Um ein neues Grafik-Objekt zu erzeugen, benötigen Sie zunächst einen selbst zu vergebenen Namen. Hinter dem Gleichheitszeichen folgen das Schlüsselwort **new** und der Aufruf der Objektfunktion **Image()**. Innerhalb der Klammern können optional zwei numerische Werte, durch ein Komma voneinander getrennt, notiert werden. Deren Bedeutung stellt sich folgendermaßen dar. Beachten Sie, dass diese Parameter bislang lediglich vom Internet Explorer interpretiert werden. Der Netscape Navigator speichert hier jeweils den Wert **0**.

mögliche Parameter

- Parameter 1 – Gibt die Breite der Grafik an.
- Parameter 2 – Gibt die Höhe der Grafik an.

Durch das Erzeugen neuer Grafik-Objekte können Bilder dynamisch ausgetauscht werden. Zur Veranschaulichung hierfür dient die nachstehende Syntax. Beim Laden der Seite wird die Grafik `Bild1.jpg` angezeigt. Fährt der Anwender mit der Maus über diese Grafik, wird diese Grafik dynamisch durch die Grafik `Bild2.jpg` ersetzt.

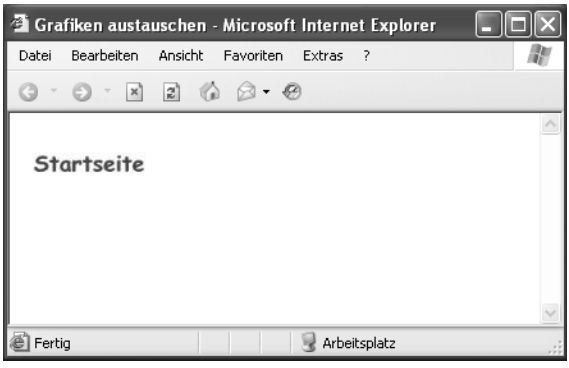


Abbildung 10.1: Zunächst wird die Grafik `Bild1.jpg` angezeigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<IMG src="Bild1.jpg" width="100" height="50" onmouseover="Bildwechsel()">
<SCRIPT type="text/JavaScript">
<!--
neueGrafik = new Image(100,50);
neueGrafik.src = "bild2.jpg";
 function Bildwechsel()
 {
  document.images[0].src = neueGrafik.src;
 }
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 10.1: Ein einmaliger Grafiktausch

Der Event-Handler **`onmouseover`** löst die Funktion `Bildwechsel()` aus. Im Script-Bereich wird ein neues Grafik-Objekt gespeichert. Als Objekt-Name wird `neueGrafik` verwendet. Zu Demonstrationszwecken werden die Breite und Höhe der neuen Grafik mit `300` mal `200` angegeben. Das neu erzeugte Grafik-Objekt kann nun weiter bearbeitet werden. Um die Quelle der Grafik zu referenzieren, wird die **`src`**-Eigenschaft verwendet. Dieser wird der Name der Grafik zugewiesen. Beachten Sie, dass Sie alle in diesem Abschnitt vorgestellten Eigenschaften auf neu erzeugte Grafik-Objekte anwenden können. Innerhalb der Funktion `Bilderwechsel()` wird nun über **`document.images[0]`** auf die erste Grafik der Datei zugegriffen. Um den Bilderwechsel zu realisieren, wird die **`scr`**-Eigenschaft verwendet. Der zunächst angezeigten Grafik wird die **`scr`**-Eigenschaft, und somit die Quelle des neuen Grafik-Objekts, zugewiesen.

Abbildung 10.2: Durch das `onmouseover`*-Event wird die Grafik Bild2.jpg angezeigt.*

10.2 Rahmen

Der Wert des `border`***-Attributs wird ausgelesen.***

Mit der Eigenschaft **`border`** lässt sich die Stärke des eine Grafik umgebenden Rahmens auslesen und abspeichern. Das hierfür verwendete HTML-Attribut, auf welches sich diese JavaScript-Anweisung bezieht, lautet **`border`**. Nach HTML-Syntax ist hierbei ein beliebig großer numerischer Wert gestattet, wobei **`0`** die Nichtanzeige des Rahmens zur Folge hat. Im folgenden Beispiel wird innerhalb des einleitenden **`<IMG>`**-Tags die Anweisung **`border`** mit einem Wert von `3` notiert. Exakt dieser Wert soll, durch den Event-Handler **`onmouseover`** ausgelöst, in einem **`alert()`**-Fenster angezeigt werden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<IMG src="erde.jpg" width="30" height="90" border="3" name="Erde"
onmouseover="alert(document.Erde.border)">
</BODY>
</HTML>
```

Listing 10.2: Der Wert `3` *wird ausgegeben.*

Um die Rahmenstärke zu ermitteln, wird in diesem Beispiel der eingebundenen Grafik ein eindeutiger Name zugewiesen. Innerhalb der **`alert()`**-Methode wird die Rahmenstärke der Grafik über **`document`**`.erde.`**`border`** ausgelesen.

10.3 Ladezustand von Grafiken überprüfen

Fehlermeldungen vermeiden

Die Eigenschaft **`complete`** speichert, ob eine Grafik innerhalb einer HTML-Seite vollständig geladen wurde. Diese Information eignet sich vorzüglich für solche Anwendungen, bei denen das tatsächliche Vor-

handensein aller Grafiken ein notwendiger Bestandteil ist. Exemplarisch hierfür sei ein grafisches Pulldown-Menü genannt. Hier wäre es fatal, wenn nicht alle Grafiken bereits zu Beginn angezeigt werden würden. Das Fatale hieran bezieht sich einerseits auf den ästhetischen Part der Anwendung, aber auch die gewünschte Funktionalität würde in Mitleidenschaft gezogen. Die Eigenschaft **complete** kennt die beiden Werte **true** und **false**. Wurde die gewünschte Grafik geladen, wird der Wert **true** zugewiesen. Tritt der gegenteilige Effekt, also die Nichtanzeige der Grafik, ein, wird der Wert **false** herangezogen. Das folgende Beispiel zeigt eine Anwendung, durch die erst dann eine Weiterleitung auf eine neue Seite erfolgt, wenn alle Grafiken vollständig geladen sind.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function geladeneGrafik(){
if (document.images[0].complete == true && document.images[1].
complete == false && document.images[2].complete == false );
location.href = "intro.htm";
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P>Die Grafiken werden geladen</P>
<IMG src="grafik1.gif" onload="geladeneGrafik()" width="1" height="1">
<IMG src="grafik2.gif" onload="geladeneGrafik()" width="1" height="1">
<IMG src="grafik3.gif" onload=" geladeneGrafik()" width="1" height="1">
</BODY>
</HTML>
```

Listing 10.3: Wurden alle Grafiken geladen?

Innerhalb der Funktion `geladeneGrafik()` wird überprüft, ob die drei innerhalb des Dateikörpers integrierten Grafiken vollständig geladen wurden. Trifft dies zu, wird der Anwender von der aktuellen Seite auf die Datei `intro.htm` geleitet. Anderenfalls findet keine automatische Weiterleitung statt. Für das Auslösen der Funktion `geladeneGrafik()` wird innerhalb jeder der drei Grafiken der Event-Handler **onload** verwendet.

10.4 Grafikhöhe

Die **heigth**-*Angabe ist nicht notwendig.*

Anhand der Eigenschaft **height** kann die Höhe einer Grafik gespeichert werden. Beachten Sie, dass auch dann eine gültige Höhe ausgelesen wird, wenn innerhalb des HTML-Tags **<IMG>** keine **height**-Angabe notiert wurde. Für den Fall, dass sich innerhalb des **<IMG>**-Tags tatsächlich eine Angabe zur Höhe der aktuellen Grafik befindet, wird diese innerhalb der Funktion vorrangig der JavaScript-Eigenschaft **height** behandelt. Die **height**-Eigenschaft sollte stets im Kontext mit der **width**-Angabe verstanden und auch stets mit dieser zusammen verwendet werden. Mehr zu der **width**-Eigenschaft erfahren Sie im Abschnitt Grafikbreite ab Seite 341. Das folgende Beispiel zeigt eine Symbiose aus der Eigenschaft **height** und dem Öffnen eines neuen Fensters. Der Sinn der hier aufgeführten Funktion besteht darin, dass mit dem Zeigen auf die integrierte Grafik ein neues Fenster geöffnet wird. Hierin wird die entsprechende Grafik in ihrer Originalgröße angezeigt. Die Vorteile dieser Funktion liegen auf der Hand. So braucht es fortan nicht mehrere Funktionen, um ein neues Fenster mit der gewünschten Größe zu öffnen. Es reicht der Einsatz einer globalen Funktion. Das Haupteinsatzgebiet der hier gezeigten Anwendung dürfte sich demzufolge vor allem auf Thumbnail-Galerien erstrecken, in denen die ausgewählten Grafiken innerhalb eines neuen Fensters angezeigt werden sollen.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<IMG src="grafik1.gif" name="Grafik"
onmouseover="neueGroesse('grafik1.gif');">
<SCRIPT type="text/JavaScript">
<!--
function neueGroesse(img)
{
var Hoehe = document.Grafik.height+30;
var Breite = document.Grafik.width+30;
oeffneFenster = this.open(img, 'Fenster', 'width=' + Breite + ',height=' +
Hoehe + ',status=no,toolbar=no,menubar=no,scrollbars=no');
oeffneFenster.focus();
}
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 10.4: Die Grafik wird in einem neuen Fenster angezeigt.

Innerhalb des Dateikörpers wird die Grafik `grafik1.gif` spezifiziert. Diese erhielt den Namen `grafik1` und wurde mit dem Event-Handler **`onmouseover`** versehen. Dieser löst die Funktion `neueGroesse()` aus. Das im Zusammenhang mit der Eigenschaft **`height`** Relevante ist hier vor allem die Variable `Hoehe`, welche sich aus der tatsächlichen Höhe der Grafik, sowie zusätzlichen `30` Pixel zusammensetzt. Der Grund für die Zugabe dieser `30` Pixel beruht darauf, dass die vollständige Grafik innerhalb des neuen Fensters angezeigt werden soll. Würde auf diesen Pixel-Zusatz verzichtet werden, könnte die Grafik innerhalb des neuen Fensters nicht vollständig angezeigt werden. Im übrigen Verlauf der Funktion `neueGroesse()` wird lediglich noch festgelegt, in welcher Darstellungsform das neu generierte Fenster angezeigt werden soll. Zusätzlich hierzu wird im letzten Schritt bestimmt, dass das neue Fenster fokussiert wird.

10.5 Grafikbreite

Die Eigenschaft **`width`** speichert die Breite einer Grafik. Sie sollte im Normalfall im Kontext mit der **`height`**-Eigenschaft verwendet werden. Der **`width`**-Wert bezieht sich auf die Angabe des **`width`**-Attributs, die innerhalb des **`<IMG>`**-Tags notiert wurde. Fehlt dieses Attribut, wird jedoch ebenfalls ein Wert ausgegeben. Es ist in jedem Fall darauf zu achten, dass im Hinblick auf einen schnelleren Seitenaufbau die **`width`**-Angabe notiert werden sollte. Mittels JavaScript lässt sich die Breite einer Grafik verändern. Einen solchen Fall beschreibt das folgende Beispiel:

*Es wird auch bei fehlender **`width`**-Angabe ein Wert ermittelt.*

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function variableBreite()
{
document.all.Bild.width = document.all.Formular.Eingabe.value;
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe" value="200">
<INPUT type="button" value="Breite &auml;ndern" onclick=" variableBreite()">
</FORM>
<IMG src="gross.jpg" id="Bild" width="200" height="304">
</BODY>
</HTML>
```

Listing 10.5: Die Grafikgröße kann verändert werden.

Innerhalb des Dateikörpers befindet sich ein Formular, in welchem ein einzeiliges Eingabefeld sowie ein Button definiert sind. Unterhalb dieses Formulars wird eine Grafik mit der dateiweit eindeutigen ID `Bild` referenziert. Durch das Anklicken des Buttons wird die Funktion `variableBreite()` aufgerufen. In dieser wird auf die Grafik `Bild` und speziell auf deren **`width`**-Eigenschaft zugegriffen. Dieser Eigenschaft wird der Wert, welcher innerhalb des einzeiligen Eingabefeldes notiert wurde, zugewiesen. Der Anwender erhält somit die Möglichkeit, die Grafik seinen Bedürfnissen entsprechend anzupassen.

10.6 Horizontaler Abstand

liefert bei fehlendem **hspace***-Attribut den Wert* **0**

Die Eigenschaft **`hspace`** speichert den horizontalen Abstand zwischen der Grafik und dem nachfolgenden Objekt. Den entsprechenden Wert erhält diese Eigenschaft von dem innerhalb des **`<IMG>`**-Tags notierten **`hspace`**-Attribut. Dieses erwartet einen ganzzahligen Wert. Der diesem Attribut zugewiesene Zahlenwert wird dabei nicht nur an einer, sondern an beiden Seiten der Grafik gleichermaßen eingehalten. Wurde innerhalb des **`<IMG>`**-Tags kein **`hspace`**-Attribut notiert oder wurde diesem kein Wert zugewiesen, liefert die JavaScript-Eigenschaft **`hspace`** den Wert **`0`**. Ein Beispiel:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:alert(document.Grafik.hspace)">
<IMG src="maske.gif" hspace="5" name="Grafik" border="0">
</A>
</BODY>
</HTML>
```

Listing 10.6: Der horizontale Abstand wird in einem Meldungsfenster angezeigt.

Innerhalb der gezeigten Syntax wird die Grafik `maske.gif` notiert und ihr der Name `Grafik` zugewiesen. Die gesamte Grafikreferenz wird innerhalb eines **`<A>`**-Tags notiert und somit als Hyperlink spezifiziert. Wird dieser ausgelöst, wird der innerhalb des **`<IMG>`**-Tags vergebene **`hspace`**-Wert ausgelesen und in einem **`alert()`**-Fenster angezeigt. In diesem Beispiel ist der ausgegebene Wert `5`.

10.7 Vertikaler Abstand

liefert bei fehlendem **vscpace***-Attribut den Wert* **0**

Mit der Eigenschaft **vspace** kann der vertikale Abstand zwischen einer Grafik und den Elementen, die sich oberhalb bzw. unterhalb von dieser befinden, gespeichert werden. Hierbei wird der Wert ausgegeben, welcher innerhalb des **<IMG>**-Tags dem Attribut **vspace** zugewiesen wurde. Gesetzt den Fall, das **vspace**-Attribut wurde innerhalb des **<IMG>**-Tags nicht notiert, wird der Wert **0** für die JavaScript-Eigenschaft gespeichert. Beachten Sie, dass sich das Attribut **vspace** stets gleichermaßen auf die Abstände oben und unten bezieht. Wurde also beispielsweise die Anweisung **vspace="**3**"** notiert, bedeutet dies, dass zwischen der Grafik und den oben und unten angrenzenden Elementen jeweils ein Abstand von 3 Pixel eingehalten wird. Den Einsatz von **vspace** in JavaScript soll die folgende Syntax veranschaulichen:

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<A href="javascript:alert(document.Grafik.vspace)">
<IMG src="maske.gif" vspace="5" name="Grafik" border="0">
</A>
</BODY>
</HTML>
```

Listing 10.7: Der vertikale Abstand wird in einem Meldungsfenster angezeigt.

Innerhalb der aufgeführten Syntax wird die Grafik mit dem Namen `Grafik` notiert. Beim Anklicken dieser Datei wird innerhalb eines **alert()**-Fensters der Wert, welcher dem **vspace**-Attribut zugewiesen wurde, in diesem Beispiel beträgt dieser `5`, ausgegeben.

10.8 Anzahl der vorhanden Grafiken

Beispiel: SlideShow

Wie viele Grafiken sich innerhalb einer HTML-Datei befinden, kann in vielerlei Hinsicht von Interesse sein. Für diesen Zweck steht die Eigenschaft **length** zur Verfügung. Die Einsatzvarianten für diese Eigenschaft sind zahlreich und reichen von dem Ausgeben der tatsächlich vorhandenen Anzahl von Grafiken innerhalb einer Seite bis hin zum Einsatz innerhalb von Scripts, welche die Darstellung einer SlideShow nach sich ziehen. Und eben einen solchen Fall beschreibt die nachfolgende Syntax. Hier werden sechs Grafiken spezifiziert, die wechselseitig innerhalb der HTML-Datei mit einer Geschwindigkeit von `2.500` Millisekunden ausgetauscht werden. Eine solche Syntax eignet sich jedoch nicht nur für den Einsatz innerhalb komplexer seitenfüllender SlideShows, son-

dern kann beispielsweise auch für die Darstellung optisch ansprechender Werbebanner genutzt werden. Die Syntax im Überblick:

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
 var Grafiken=new Array('grafik.gif',
           'grafik1.gif',
           'grafik2.gif',
           'grafik3.gif',
           'grafik4.gif',
           'grafik5.gif');
 var Geschwindigkeit=2500;
 var Beginn=0;
function Rotation () {
if (!(document.images)) {return;}
 document.leer.src=Grafiken[Beginn++];
 if (Beginn == Grafiken.length) { Beginn = 0; }
 setTimeout("Rotation();",Geschwindigkeit);
}
//-->
</SCRIPT>
</HEAD>
<BODY onload="Rotation();">
<IMG src="leer.gif" name="leer" width="200" height="100">
</BODY>
</HTML>
```

Listing 10.8: Die Realisierung einer SlideShow

Mit dem Aufrufen der gezeigten HTML-Datei wird die Funktion `Rotation()` aufgerufen. Innerhalb dieser Funktion werden die drei Variablen `Grafiken` (als Array), `Geschwindigkeit` sowie `Beginn` definiert. Beachten Sie, dass die Grafik `leer.gif` innerhalb des Dateikörpers lediglich als Platzhalter dient.

10.9 Vorschaugrafik

Fehlermeldungen verhindern

Innerhalb des **<IMG>**-Tags kann per HTML-Syntax und mit dem Attribut **lowsrc** eine Vorschaugrafik spezifiziert werden. Hierbei handelt es sich im Normalfall um eine Grafik, die mit einer geringen Auflösung abgespeichert wurde und somit zwar qualitativ nicht hochwertig ist, dafür jedoch schnellere Ladezeiten ermöglicht. Die JavaScript-Methode **lowsrc** gestattet, die dem Attribut **lowsrc** zugewiesene Grafik auszulesen. Im Hinblick auf JavaScript-Anwendungen eignet sich die **lowsrc**-Eigen-

schaft beispielsweise für den Einsatz innerhalb von Bilderwechseln. Wie sich die **lowsrc**-Eigenschaft innerhalb einer solchen Anwendung nutzen lässt, beschreibt das folgende Beispiel. Innerhalb des Dateikörpers befindet sich ein HTML-Formular, in welchem drei Checkboxen definiert sind. Jeder dieser Checkboxen werden zwei Grafiken zugewiesen – jeweils eine mit geringerer Auflösung sowie die Originalgrafik. Durch das Anklicken einer der Checkboxen wird die Funktion `Bildwechsel()` aufgerufen. Diese definiert, dass zunächst immer die **lowsrc**-Grafik und anschließend die Originalgrafik angezeigt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Bildwechsel(niedrig,normal)
{
document.images[0].lowsrc=niedrig;
document.images[0].src=normal;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="formular">
Bild1:<INPUT type="radio" name="grafik" value="grafik1"
onclick="Bildwechsel('grafik1.gif','grafik2.gif')">
Bild2:<INPUT type="radio" name="grafik" value="grafik2"
onclick="Bildwechsel('grafik3.gif','grafik4.gif')">
Bild3:<INPUT type="radio" name="grafik" value="grafik3"
onclick="Bildwechsel('grafik5.gif','grafik6.gif')">
<BR>
<IMG src="grafik2.gif" name="grafiken" lowsrc="grafik1.gif">
</FORM>
</BODY>
</HTML>
```

Listing 10.9: Der Anwender bekommt sehr schnell etwas angezeigt.

Beachten Sie, dass sich die aufgeführte Syntax nur dann als tatsächlich praxistauglich erweist, wenn die Unterschiede zwischen der **lowsrc**- und der Originalgrafik auch vorhanden sind und durch den Anwender nachvollzogen werden können. Größenunterschiede von 1-2 KByte rechtfertigen hingegen kaum den Einsatz einer über **lowsrc** eingefügten Vorab-Grafik.

10.10 Name

Es wird auf den Wert des **name***-Attributs zugegriffen.*

Die Eigenschaft **name** ermöglicht es, den Namen der entsprechenden Grafik auszulesen. Hierbei greift JavaScript auf das innerhalb des HTML-Tags **<IMG>** zugewiesene Attribut **name**, genauer gesagt, dessen Wert, zu. Beachten Sie, dass dieser Name weder Sonderzeichen noch Umlaute enthalten sollte. Innerhalb der Browser konnte bislang der Wert von **name** lediglich ausgelesen werden. Dieses Verhalten hat sich nun, zumindest innerhalb des Netscape Navigators ab Version 6, geändert. Hier lässt sich der vergebene Name auch ändern.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Ausgabe()
{
  alert(document.MeineGrafik.name);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<IMG name="MeineGrafik" src="uhr.gif">
<BR>
<A href="javascript:Ausgabe()">Zeige Namen</A>
</BODY>
</HTML>
```

Listing 10.10: Der Grafikname wird ausgegeben.

Innerhalb des gezeigten Beispiels wird die Grafik `uhr.gif` referenziert. Dieser wird über das entsprechende HTML-Attribut der Name `MeineGrafik` zugewiesen. Unterhalb dieser Grafik befindet sich ein Verweis. Ein Klick darauf ruft Funktion `Ausgabe()` auf. Diese Funktion bewirkt, dass innerhalb eines **alert()**-Fensters der Name der eingebundenen Grafik angezeigt wird.

10.11 Quelle der Grafik

Voraussetzung für Bilderwechsel

Über die **src**-Eigenschaft wird gespeichert, welche Grafik angezeigt wird bzw. werden soll. Die **src**-Eigenschaft bezieht sich hierbei auf den Wert des **src**-Attributs innerhalb des **<IMG>**-Tags. Da sich der Wert der **src**-Eigenschaft nicht nur lesen, sondern auch ändern lässt, wird der dynamische Bildtausch möglich. Vor der Realisierung eines solchen Bilderwechsels sollten Sie bedenken, dass die innerhalb des **<IMG>**-Tags notierte

Bildgröße auf alle über die **src**-Eigenschaft neu angezeigten Grafiken angewandt wird. Um also etwaige Verzerrungen zu vermeiden, sollten alle wechselseitig anzuzeigenden Grafiken die gleiche Größe besitzen. Das folgende Beispiel zeigt, wie sich Grafiken austauschen lassen. Hier werden die Grafiken jedoch nicht automatisch verändert. Vielmehr kann der Anwender einen Listeneintrag auswählen und die sich hinter diesem Eintrag verbergende Grafik wird anschließend unterhalb der Auswahlliste angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Bildwechsel()
{
for(i=0; i<document.Auswahl.Waehler.length; i++)
 if(document.Auswahl.Waehler.options[i].selected == true)
  document.images.Grafiken.src = document.Auswahl.Waehler[i].text;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Auswahl">
<SELECT name="Waehler" onclick="Bildwechsel()" size="1">
<OPTION>Bild1.jpg</OPTION>
<OPTION>Bild2.jpg</OPTION>
<OPTION>Bild3.jpg</OPTION>
</SELECT>
</FORM>
<IMG src="Bild1.jpg" width="100" height="50" name="Grafiken">
</BODY>
</HTML>
```

Listing 10.11: Ein Grafikwechsel durch eine Auswahlliste

Die Auswahlliste ist mit dem Event-Handler **onclick** versehen. Dessen Aktivierung ruft die Funktion `Bildwechsel()` auf. Dieser überprüft zunächst mittels einer **if**-Abfrage, welcher Listeneintrag selektiert wurde. Im nächsten Schritt wird über **document.images.**`Grafiken` auf die sich innerhalb der Datei befindliche Grafikreferenz zugegriffen. Deren **src**-Eigenschaft wird nun mit dem Text des selektierten Listeneintrags gleichgesetzt.

10.12 Grafiken wechseln

Verweise werden innerhalb von WWW-Projekten häufig als Grafiken realisiert. JavaScript bietet die Möglichkeit, solche Verweise optisch aufzuwerten. Häufig wird hierzu eine Grafik angezeigt, die sich beim Überfahren mit der Maus in eine andere Grafik verwandelt. Selbstverständlich verwandelt sich nicht die Grafik, sondern die **src**-Eigenschaft des **images**-Objekts wird verändert. In diesem Abschnitt sehen Sie, wie sich Grafiken dynamisch und in Abhängigkeit von einem Ereignis austauschen lassen.

Abbildung 10.3: Der Verweis im Normalzustand

Abbildung 10.3 zeigt die Seite nach dem Laden. Angezeigt wird hier die Grafik `Bild1.jpg`. Diese soll nun durch das Auftreten des Ereignisses **onmousover** mit einer anderen Grafik vertauscht werden. Um einen solchen Effekt zu erzielen, ist es wichtig, dass Sie zwei verschiedene Grafiken einsetzen. Um unschöne Verschiebungen zu vermeiden, sollten beide Grafiken gleich groß sein. Nach dem Überfahren der Grafik mit der Maus ergibt sich eine Darstellung wie in Abbildung 10.4.

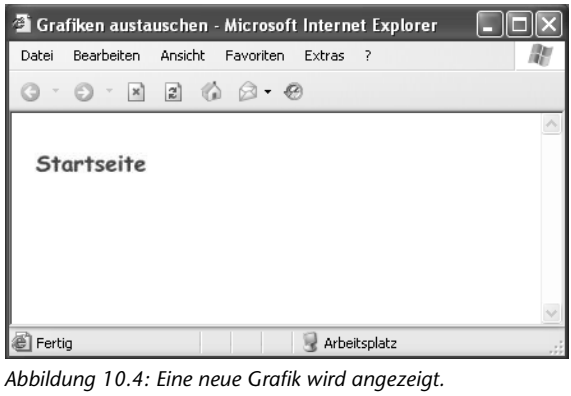


Abbildung 10.4: Eine neue Grafik wird angezeigt.

Die Funktion ist so konzipiert, dass der Grafikwechsel nicht nur einmal stattfindet. Immer, wenn der Event-Handler **onmousover** eintritt, wird die

Grafik `Bild2.jpg` angezeigt. Wird die Grafik mit der Maus verlassen, tritt also das Ereignis **onmouseout** ein, wird die Grafik `Bild1.jpg` angezeigt. Beachten Sie, dass dieses Beispiel so ausgelegt ist, dass mehrere Grafiken eingebunden werden können. Geeignet ist es somit für die Darstellung komplexer Menüstrukturen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
Normalzustand = new Image();
Normalzustand.src = "Bild1.jpg";
Effektzustand = new Image();
Effektzustand.src = "Bild2.jpg";
function Grafikwechsel(Grafiknummer,Grafikname)
{
 window.document.images[Grafiknummer].src = Grafikname.src;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="index.htm"
   onmouseover="Grafikwechsel(0,Effektzustand)"
   onmouseout="Grafikwechsel(0,Normalzustand)"><img src="Bild1.jpg"
   width="100" height="50" border="0"></A>
</BODY>
</HTML>
```

Listing 10.12: Ein einfacher Grafikwechsel

Sie müssen für beide Grafiken eine Instanz des **image**-Objekts erzeugen. Im Beispiel werden somit die beiden neuen Instanzen `Normalzustand` und `Effektzustand` generiert. Durch die Notation der **src**-Eigenschaft kann beiden Grafikobjekten eine Grafikdatei zugewiesen werden. Beachten Sie, dass Sie hierbei die kompletten Pfade der Grafiken angeben müssen. Im gezeigten Beispiel befinden sich beide Grafiken in dem gleichen Verzeichnis wie die HTML-Datei. Da diese Anweisungen nicht in eine Funktion eingebettet sind, werden sie direkt mit dem Einlesen der Datei ausgeführt. Für den Anwender ist dies jedoch zunächst nicht sichtbar. Visuell geschieht erst etwas, wenn der Anwender mit der Maus über die Grafik fährt und wenn er diese wieder verlässt. In beiden Fällen wird die Funktion `Grafikwechsel()` aufgerufen. Die Funktion benötigt zwei Parameter. Der erste Parameter gibt an, die wievielte Grafik der Datei vertauscht werden soll. `0` steht für die erste, `1` für die zweite usw. auszutauschende Grafik. Der zweite Parameter gibt an, durch welche Grafik die

aktuell angezeigte ersetzt werden soll. Der Aufruf der Funktion `Grafikwechsel()` wird mittels der beiden Event-Handler **`onmousover`** und **`onmouseout`** realisiert. Der Funktion werden jeweils die beiden Parameter übergeben.

Wie bereits erwähnt, ist die Funktion so angelegt, dass Sie mehrere Grafiken dynamisch austauschen können. Hierzu müssen Sie lediglich weitere Instanzen des **`image`**-Objekts erzeugen. Diese könnten beispielsweise *`Normalzustand1`* und *`Effektzustand1`* heißen. Beim Aufruf müssen dann als Parameter die Indexnummer sowie der Objektname übergeben werden.

10.13 Fliegende Grafik

Browserunterschiede berücksichtigen

Mit relativ geringem Aufwand lassen sich Grafiken so animieren, dass sie in vorgegebener Weise durch das Browserfenster fliegen. Im folgenden Beispiel bewegt sich die Grafik *`bewegung.gif`* entlang einer vordefinierten Kurve. Einige Vorüberlegungen: Um die Grafik zu animieren, wird diese in einen **`<DIV>`**-Bereich eingebettet. Im nächsten Schritt müssen wir uns darüber im Klaren sein, dass eine solche Anwendung für die verschiedenen Browser auch unterschiedlich geschrieben werden muss. Unser Beispiel beschränkt sich hierbei auf den Internet Explorer und den Netscape Navigator in den Produktversionen ab 4.x.

```
<HTML>
<HEAD>
<STYLE type="text/css">
.Bereich{position:absolute;font-size:18pt;font-weight:bold;color:red;}
</STYLE>
<SCRIPT type="text/JavaScript">
<!--
 var x=0;
 var Browser=navigator.appName.charAt(0);
 function Bewegung()
 {
 x+=2; if(x>=850)x=0;
 y=105-100*Math.cos(3.65+x/50);
if(Browser=="M")
{
sprunghaft.style.top=y;
sprunghaft.style.left=x;
}
else
{
document.layers['sprunghaft'].moveTo (x, y);
}
```

```
window.setTimeout('Bewegung()',25); }
//-->
 </SCRIPT>
 </HEAD>
 <BODY onload="Bewegung()">
 <DIV id="sprunghaft" name="sprunghaft" class="Bereich">
<IMG src="bewegung.gif"  height="100" width="50"></DIV>
</BODY>
</HTML>
```

Listing 10.13: Die eingebundene Grafik bewegt sich über den Bildschirm.

Die Funktion `Bewegung()` wird durch den Event-Handler **`onload`** aufgerufen. Innerhalb der Variablen `Browser` wird mittels der **`appName`**-Eigenschaft des **`navigator`**-Objekts der Name des verwendeten Browsers ermittelt. Durch die Anweisung **`charAt(0)`** wird der erste Buchstabe des Rückgabewertes gespeichert. Diesen Wert benötigen wir für unsere kleine Browserweiche. Die Funktion `Bewegung()` legt die Art der Grafikbewegung fest, wobei beispielsweise der Wert `850` die Breite der Bewegung definiert. `850` legt also fest, dass sich die Grafikbewegung über eine Breite von `850` Pixel erstreckt. Eine Aussage über den eigentlichen Verlauf der Kurve wird allerdings erst durch die Deklaration der Variablen `x` vorgenommen. Die im Beispiel verwendeten Werte können Sie Ihren Wünschen entsprechend anpassen. Durch die **`if`**-Abfrage wird überprüft, ob der Rückgabewert der Variablen `Browser` gleich `M` ist. Ist dies der Fall, handelt es sich um den Internet Explorer. Hier werden dann die Startpositionen des **`<DIV>`**-Bereichs von oben (durch **`top`**) und von links (durch **`left`**) festgelegt. Der **`else`**-Zweig kommt zur Anwendung, wenn der eingesetzte Browser das **`layers`**-Objekt kennt, es sich also um den Netscape Navigator handelt. In diesem Fall wird die **`moveTo()`**-Methode angewandt. Diese bewirkt ein Verschieben des **`<DIV>`**-Bereichs an eine bestimmte Position. Durch die **`setTimeout()`**-Methode wird festgelegt, dass die Funktion `Bewegung()` alle `25` Millisekunden neu aufgerufen wird. Auch dieser Wert kann variabel gewählt werden, wobei ein höherer Wert eine langsamere Bewegung der Grafik zur Folge hat.

10.14 Tageszeitabhängige Grafiken

Script-Ausbau möglich!

Die regelmäßige Aktualisierung eines Internetauftritts kann entscheidend zu dessen Erfolg beitragen und den Anwender „bei Laune halten“. Was aber, wenn für ständige Neuerungen keine Zeit bleibt? Die Frage ist recht einfach zu beantworten: Lassen Sie den Anwender in dem Glauben, dass die Seite ständig aktualisiert wird. Das lässt sich nun auf unterschiedliche Weise realisieren. Im folgenden Beispiel wird je nach Tageszeit eine andere Grafik angezeigt. Der Anwender bekommt vor 20 Uhr

eine andere Grafik zu Gesicht als nach 20 Uhr. Selbstverständlich lässt sich dieses Beispiel leicht modifizieren. So ließe sich z.B. das verwendete Zeitintervall verkürzen. Sich stündlich ändernde Grafiken wären somit auch möglich. Zudem könnten statt Grafiken auch Texte oder andere Elemente ausgetauscht werden. Betrachten Sie die nachfolgende Syntax daher als Ausgangspunkt für Ihre eigene Anwendung.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
var Datum= new Date()
var Zeit=Datum.getHours()
if (Zeit<=20)
document.write("<IMG src='tag.gif'>")
else
document.write("<IMG src='abend.gif'>")
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 10.14: Zwei verschiedene Grafiken für morgens und abends

Die Variable `Datum` erzeugt über **`new Date()`** ein neues Datumsobjekt. Durch die Anweisung `Datum`**`.getHours`** werden die Stunden der Tageszeit ermittelt. Der Rückgabewert wird in der Variablen `Zeit` gespeichert. Mittels einer **`if`**-Abfrage wird überprüft, ob der ermittelte Stundenwert kleiner oder gleich `20` ist. Ist diese Bedingung erfüllt, wird die Grafik `tag.gif` mittels der **`write()`**-Methode in das Dokument geschrieben. Der **`else`**-Zweig, durch den die Grafik `abend.gif` eingebunden wird, kommt dann zur Ausführung, wenn der Stundenwert größer als `20` ist.

10.15 Tagesabhängige Grafiken anzeigen

Aktualisierung der Seite wird suggeriert

In dem hier vorgestellten Beispiel geht es um Folgendes: Es wird angenommen, dass es sich bei der aufgeführten Datei um die Startseite eines Internetauftritts handelt. Um tägliche Modifikationen der Seite zu suggerieren, soll an jedem Tag der Woche eine andere Grafik angezeigt werden. Da die Woche bekanntermaßen sieben Tage hat, müssen auch sieben Grafiken vorhanden sein. Um auf das übrige Layout der Seite Rücksicht zu nehmen, sollten alle Grafiken die gleiche Größe besitzen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function taeglicheGrafik()
{
Grafiken = new Array("image0.gif", "image1.gif", "image2.gif",
"image3.gif","image4.gif", "image5.gif", "image6.gif");
Datum = new Date();
Tag = Datum.getDay();
if(document.Bild)
{
document.Bild.src=Grafiken[Tag]
}
}
 // -->
 </SCRIPT>
</HEAD>
<BODY onload="taeglicheGrafik()">
<IMG src="image1.gif" width="200" height="100" name="Bild">
</BODY>
</HTML>
```

Listing 10.15: An jedem Wochentag wird eine andere Grafik angezeigt.

Für die sieben Grafiken wird das Array `Grafiken` definiert. Dieses beinhaltet die Grafiknamen. Beachten Sie, dass sich die Grafiken hier in dem gleichen Verzeichnis wie die HTML-Datei befinden. Ist dies nicht der Fall, müssten innerhalb des Arrays die Pfadangaben angepasst werden. Über die Variable `Datum` wird ein neues Datumsobjekt mit dem aktuellen Zeitpunkt erzeugt. Durch die Anweisung `Datum.getDay()` wird der Wochentag ermittelt. Der zurückgelieferte Wert wird in der Variablen `Tag` gespeichert. Bei dem Rückgabewert handelt es sich um eine Zahl zwischen *0* und *6*, wobei *0* für Sonntag, *1* für Montag usw. stehen. Dieser Wert wird nun der **src**-Eigenschaft der Grafik zugewiesen. Ist der aktuelle Tag also gerade Donnerstag, wird der **src**-Eigenschaft der Wert `Grafiken[4]` zugewiesen. Die anzuzeigende Grafik ist in diesem Fall also `image4.gif`.

10.16 Fragen und Übungen

1. Schreiben Sie ein Programm, durch das ein Grafikwechsel realisiert werden kann. Die Auswahl der Grafiken soll aus einer Auswahlliste möglich sein. Konzipieren Sie das Programm so, dass die Grafiken unterhalb der Auswahlliste angezeigt werden. Ausgelöst wird der Bilderwechsel durch den Event-Handler `onchange`.

2. SlideShows ermöglichen das wechselseitige Anzeigen unterschiedlicher Grafiken nach einem festgelegten zeitlichen Zyklus. Eingesetzt wird dieses Stilmittel beispielsweise bei Präsentationen oder als ein netter Effekt auf Internetseiten. Um eine SlideShow zu realisieren, sollten alle einzubindenden Grafiken in der gleichen Größe vorliegen. Hierdurch lassen sich stilistisch saubere Präsentationen erstellen. In unserem Beispiel sollen vier Grafiken in einer Geschwindigkeit von 2.000 Millisekunden wechselseitig angezeigt werden. Bereits beim Laden der Seite soll eine der vier Grafiken präsentiert werden. Tipp: Legen Sie alle Grafiken in einem Array ab. Verwenden Sie die `setTimeout()`-Methode, um die entsprechende JavaScript-Funktion alle 2.000 Millisekunden neu aufzurufen.

11 Datum und Uhrzeit

lernen

Ziele dieses Kapitels

In diesem Kapitel lernen Sie den Umgang mit Datums- und Uhrzeitangaben in JavaScript. Sie erhalten hiermit die Möglichkeit, zeitabhängige Anwendungen zu erstellen. So wird es Ihnen beispielsweise möglich sein, den Anwender in Abhängigkeit von der aktuellen Uhrzeit zu begrüßen. Beachten Sie, dass sich alle per JavaScript abgefragten Uhrzeiten auf die Zeit des Anwenderrechners beziehen. Sie können somit beispielsweise ermitteln, welcher Tag in der Systemzeit des Anwenders gerade ist. Der tatsächliche Tag (der wirklichen Welt) lässt sich hiermit jedoch nicht erfragen.

11.1 Zugriff auf Datum und Uhrzeit

Um auf die Uhrzeit oder das Datum des Anwenderrechners zugreifen zu können, muss das **date**-Objekt verwendet werden. Bei JavaScript spielt der 1. Januar 1979 eine entscheidende Rolle. Wie in anderen Programmiersprachen und der EDV allgemein wird auch in JavaScript ein fester Punkt für Datumsberechnungen benötigt. Und in JavaScript ist dies eben der 1. Januar 1979 um 0.00 Uhr. Um auf Datums- und Uhrzeitfunktionen Zugriff zu bekommen, muss zunächst ein neues **Date()**-Objekt erzeugt werden. Hierfür existieren verschiedene Syntaxformen.

```
Objektname = new Date();
```

Um die aktuelle Uhrzeit und das Datum zu speichern, wird hinter dem frei wählbaren Objektnamen über **new Date()** eine neue Objektinstanz erzeugt. Soll das neue Datumsobjekt mit einem bestimmten Zeitpunkt initialisiert werden, stehen die drei folgenden Varianten zur Verfügung:

```
zeigeJahr = new Date("september 29, 2002 10:29:55");
```

Hierbei muss der Name des Monats in englischer Schreibweise angegeben werden. Nicht durch Komma getrennt, folgt der Monatstag als numerischer Wert. Hieran schließen sich das Jahr, die Stunden, Minuten und Sekunden an.

```
zeigeJahr = new Date(2002,8,29);
```

Hierdurch werden das Jahr, der Monat sowie der Monatstag mit dem gewünschten Wert initialisiert. Beim Monat ist darauf zu achten, dass die interne Zählung bei 0 beginnt. Der Januar hat somit den Wert 0, der Februar den Wert 1 usw.

```
zeigeJahr = new Date(2002,8,29,10,29,30);
```

Durch diese Variante werden das Jahr, der Monat, der Monatstag, die Stunden, Minuten und Sekunden des Datumsobjekts initialisiert. Für den Monat gilt wieder, dass die interne Zählung bei 0 beginnt. Die in dem Beispiel verwendete `8` beschreibt somit den September.

11.2 Monatstag ermitteln

Werte zwischen **1** *und* **31**

Die Methode **`getDate()`** liefert den Tag des aktuellen Datums zurück. Der hierbei zu erwartende Rückgabewert liegt zwischen **1** und **31**. Wurde also beispielsweise innerhalb des Objekts das Datum 10.8.2002 gespeichert, wird 10 zurückgegeben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
var neuesDatum = new Date();
var Tag = neuesDatum.getDate();
document.write(Tag);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.1: Der aktuelle Tag wird in das Dokument geschrieben.

In dem gezeigten Beispiel wird über **`new Date()`** ein neues Datumsobjekt mit dem Objektnamen `neuesDatum` erzeugt. Der Variablen `Tag` wird über `neuesDatum`**`.getDate()`** der aktuelle Monatstag zugewiesen und anschließend über die **`write()`**-Methode dynamisch in die Datei geschrieben.

11.3 Wochentag ermitteln

Werte zwischen 1 und 6

Mittels der **getDay()**-Methode wird der Wochentag der lokalen Zeit gespeichert. Hierbei werden numerische Werte zwischen **0** und **6** zurückgeliefert. Wobei **0** für Sonntag, **1** für Montag usw. stehen. So würde also für den 31.10.2002 der Wert 4 ausgegeben werden, da dieser ein Donnerstag war. Da sich die Ausgabe eines numerischen Wertes nur bedingt zur Darstellung des Wochentags eignet, sollte dieser Wert umgewandelt werden. Wie sich dies realisieren lässt, beschreibt die nachstehende Syntax.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Wochentag()
{
var neuesDatum = new Date();
var aktuellerWochentag = neuesDatum.getDay();
var Tage = new
Array("Sonntag","Montag","Dienstag","Mittwoch","Donnerstag","Freitag",
"Samstag");
alert(Tage[aktuellerWochentag]);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Wochentag()">Welcher Tag ist heute?</A>
</BODY>
</HTML>
```

Listing 11.2: Die Ausgabe des Wochentags in einem Meldungsfenster

Zunächst wird in dem Objektnamen `neuesDatum` ein neues Datumsobjekt erstellt. Die Variable `aktuellerWochentag` speichert den numerischen Wert des Wochentags. Um nicht diesen Wert, sondern einen tatsächlich lesbaren Wochentag auszugeben, wird ein Array mit den Namen der entsprechenden Wochentage erzeugt. Um eine Übereinstimmung zwischen der **getDay()**-Methode und den Wochentagsnamen des Arrays zu erreichen, ist darauf zu achten, dass das erste Element der Sonntag sein muss. Abschließend wird der ermittelte Wert durch die **alert()**-Methode in einem Meldungsfenster ausgegeben.

Abbildung 11.1: Der aktuelle Wochentag wird in einem Meldungsfenster angezeigt.

11.4 Volles Jahr ermitteln

Die **getFullYear()**-Methode liefert die Jahreszahl eines **Date**-Objekts zurück. Im nachstehenden Beispiel führt die Verwendung dieser Methode dazu, dass aus dem aktuellen Zeitpunkt lediglich der Wert 2002 ausgegeben wird. Dieser Wert wird anschließend in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Jahreszahl()
{
zeigeJahr = new Date( "2002", "08", "13", "8","20" );
alert( "Jahr: " + zeigeJahr.getFullYear() );
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Jahreszahl()">Welches Jahr?</A>
</BODY>
</HTML>
```

Listing 11.3: Nur die Jahreszahl wird ausgegeben.

Durch Anklicken des Verweises wird die Funktion Jahreszahl() ausgeführt. Hierin wird zunächst ein neues Datumsobjekt mit einem bestimmten Zeitpunkt notiert. Mittels der **getFullYear()**-Methode wird aus diesem **Date**-Objekt die vorhandene Jahreszahl gespeichert und mittels der **alert()**-Methode ausgegeben.

11.5 Stunden ermitteln

Mittels der **getHours()**-Methode kann die Stunde der aktuellen Uhrzeit ermittelt werden. Ist die aktuelle Uhrzeit also beispielsweise 12.23.00, wird der Wer 12 zurückgeliefert. Im nachfolgenden Beispiel werden die Stunden und Minuten der lokalen Zeit über die **write()**-Methode dynamisch in das Dokument geschrieben.

```
<HTML>
<BODY>
<P>Akutell ist es:
<SCRIPT type="text/JavaScript">
var Zeit=new Date();
document.write(Zeit.getHours() + ":" + Zeit.getMinutes());
```

```
</SCRIPT>
</P>
</BODY>
</HTML>
```

Listing 11.4: Die Stunde der Uhrzeit wird in das Dokument geschrieben.

Zunächst wird innerhalb des Script-Bereichs ein neues Datumsobjekt mit dem Namen `Zeit` erzeugt. Anschließend werden die beiden Methoden **`getHours()`** und **`getMinutes()`** notiert. Dies hat die Ausgabe der aktuellen Stunde und Minuten der lokalen Zeit zur Folge.

11.6 Millisekunden ermitteln

Werte zwischen ***0*** *und* ***999***

Die **`getMilliseconds()`**-Methode liefert die Anzahl der Millisekunden, die seit der letzten vollen Sekunden vergangen sind. Hierbei werden Werte zwischen **`0`** und **`999`** zurückgeliefert. Anhand der folgenden Syntax kann sich der Anwender die seit der letzten vollen Sekunde vergangenen Millisekunden mittels Button-Klick anzeigen lassen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function vergangeneZeit()
{
Zeit = new Date();
Ausgabe = Zeit.getMilliseconds();
document.write(Ausgabe);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM>
<INPUT type="button" value="Anzeigen" onclick="vergangeneZeit()">
</FORM>
</BODY>
</HTML>
```

Listing 11.5: Die Ausgabe der vergangenen Millisekunden

Durch Anklicken des Formular-Buttons wird die Funktion `vergangeneZeit()` ausgelöst. Innerhalb dieser Funktion wird zunächst ein neues Datumsobjekt mit dem Namen `Zeit` erzeugt. Über die **`getMilliseconds()`**-Methode werden innerhalb der Variablen `Ausgabe` die seit der letzten Sekunde vergangenen Millisekunden gespeichert. Der ermittelte Wert wird anschließend über die **`alert()`**-Methode ausgegeben.

11.7 Minuten ermitteln

Die **getMinutes()**-Methode liefert die Minuten der aktuellen Uhrzeit zurück. Wurde beispielsweise die Uhrzeit 10.45.29 gespeichert, so wird hier der Wert 45 zurückgeliefert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var aktuelleMinuten = new Date();
alert(aktuelleMinuten.getMinutes());
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 11.6: Die Minuten der aktuellen Uhrzeit werden ausgegeben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgeführt. Im Beispiel wird ein neues Datumsobjekt namens `aktuelleMinuten` erzeugt. Durch `aktuelleMinuten`**.getMinutes()** werden die Minuten des aktuellen Datums ermittelt. Der ausgelesene Wert wird mittels der **alert()**-Methode ausgegeben.

11.8 Monat ermitteln

Werte zwischen 0 und 11

Über die **getMonth()**-Methode kann der Monat ermittelt werden. So würde bei der Verwendung des Datums 4.11.2002 der Wert 10 zurückgeliefert. Dass hier nicht der Wert 11 ausgegeben wird, liegt daran, dass die Zählung der Monate bei **0** beginnt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
var neuesDatum = new Date();
var Monat = neuesDatum.getMonth();
var Monatsname = new
Array("Januar","Feb-
ruar","M&auml;rz","April","Mai","Juni","Juli","August","September","Okto-
ber","November","Dezember");
```

```
alert(Monatsname[Monat]);
//-->
</SCRIPT>
<BODY>
</BODY>
</HTML>
```

Listing 11.7: Der Monat wird ausgegeben.

Im aufgeführten Beispiel wird zunächst ein neues Datumsobjekt namens `neuesDatum` erzeugt. Über `neuesDatum.getMonth()` wird der numerische Wert des Monats ermittelt. Dieser Wert selbst ist nun wenig aussagekräftig. Aus diesem Grund soll dieser Wert dazu verwendet werden, den Monatsnamen zu ermitteln. Hierzu bedient sich die Syntax eines Arrays. Innerhalb des Arrays werden alle Monatsnamen aufgeführt. Beachten Sie, dass auch bei Arrays die interne Zählung bei **0** beginnt. Durch die Anweisung `Monatsname[Monat]` wird der ermittelte Wert des Datumsobjekts `Monat` mit dem Array verglichchen. Statt des numerischen Wertes wird hierdurch der Monatsname durch die Verwendung der **alert()**-Methode ausgegeben.

Abbildung 11.2: Der aktuelle Monat wird angezeigt.

11.9 Sekunden ermitteln

Die **getSeconds()**-Methode liefert die Sekunden der aktuellen Uhrzeit. Wurde im aktuellen Datumsobjekt die Zeit 20.59.12 gespeichert, so wird hier der Wert 12 zurückgeliefert. Im nachstehenden Beispiel werden die Sekunden der aktuellen Uhrzeit dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<P>Die Sekunden vergehen
<SCRIPT type="text/JavaScript">
var neuesDatum=new Date();
document.write(neuesDatum.getSeconds());
</SCRIPT>
</P>
<INPUT type="button" onclick="javascript:location.reload()"
```

```
value="aktualisieren">
</BODY>
</HTML>
```

Listing 11.8: Die Sekunden werden dynamisch in das Dokument geschrieben.

Der Script-Bereich wird mit dem Einlesen der Datei ausgeführt. Hierin wird zunächst das neue Datumsobjekt `neuesDatum` erstellt. Durch `neuesDatum.getSeconds()` werden die aktuellen Sekunden ermittelt und über die **`write()`**-Methode in das Dokument geschrieben. Um den ermittelten Wert aktualisieren zu können, befindet sich innerhalb des Dateikörpers ein Formular-Button. Durch die **`reload()`**-Methode des **`location`**-Objekts wird die Seite bei jedem Anklicken des Buttons neu geladen und der Sekundenwert neu ermittelt.

11.10 Zeitpunkt ermitteln

Die **`getTime()`**-Methode liefert eine numerische Darstellung der Uhrzeit zurück. Hierbei handelt es sich um die Anzahl der seit dem 1. Januar 1970 um 0 Uhr verstrichenen Millisekunden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
 var neuesDatum = new Date();
 alert(neuesDatum.getTime());
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.9: Die vergangenen Millisekunden werden in einem Meldungsfenster angezeigt.

Die aufgeführte Syntax beschreibt die Verwendung der **`getTime()`**-Methode. Zunächst wird über **`new Date()`** ein neues Datumsobjekt namens `neuesDatum` erzeugt. Dieses wird anschließend über `neuesDatum.getTime()` mit der **`alert()`**-Methode in einem Meldungsfenster ausgegeben.

11.11 Jahr ermitteln

Die **getYear()**-Methode liefert als Rückgabewert die Jahreszahl des aktuellen Datums. Ist das aktuelle Datum beispielsweise der 9.10.2002, wird der Wert 2002 gespeichert. Im nachstehenden Beispiel wird die Jahreszahl ermittelt und in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
var neuesDatum = new Date();
var Jahr = neuesDatum.getYear();
alert(Jahr);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.10: Das aktuelle Jahr wird in einem Meldungsfenster ausgegeben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgelöst. Hierin wird zunächst das neue Datumsobjekt `neuesDatum` erzeugt. Dieses Objekt sowie die **getYear()**-Methode werden der Variablen `Jahr` zugewiesen. Der ermittelte Wert dieser Variablen wird anschließend über die **alert()**-Methode ausgegeben.

11.12 Millisekunden seit dem 1.1.1970 ermitteln

Die **parse()**-Methode ermittelt die Anzahl der zwischen dem 1.1.1970 um 0:00:00 Uhr und dem übergebenen Datum verstrichenen Millisekunden. Um mit der **parse()**-Methode arbeiten zu können, muss kein neues Datumsobjekt erzeugt werden. Vielmehr genügt hier die Anweisung **Date**. Die nachstehende Syntax bewirkt, dass die verstrichenen Millisekunden dynamisch in das Dokument geschrieben werden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
Datum = "March 12, 20003"
document.write( Date.parse(Datum) );
```

```
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.11: Die vergangenen Millisekunden werden in das Dokument geschrieben.

Ausgelöst wird der Script-Bereich direkt mit dem Einlesen der Datei. Der Variablen `Datum` wird ein Datum zugewiesen. Innerhalb der **`write()`**-Anweisung wird **`Date,`** gefolgt von der **`parse()`**-Methode, notiert. Als Parameter wird dieser die Variable `Datum` zugewiesen.

11.13 Monatstag setzen

Werte zwischen 1 und 31

Die **`setDate()`**-Methode ermöglicht das Ändern eines innerhalb eines Datumsobjekts gespeicherten Monatstags. Beachten Sie, dass der neue Wert einer zwischen **1** und **31** sein sollte. Die nachfolgende Syntax bewirkt, dass der aktuelle Monatstag `10` auf `23` geändert wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Jahreszahl()
{
neuesDatum = new Date("2002", "08", "10", "1","50", "31", "49");
alert( "Datum: " + neuesDatum.getDate() );
neuesDatum.setDate( "23" );
alert( "Datum: " + neuesDatum );
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:Jahreszahl()">&Auml;ndere Monatstag</A>
</BODY>
</HTML>
```

Listing 11.12: Der alte und neue Monatstag werden ausgegeben.

Die `Jahreszahl()`-Funktion wird durch Anklicken des Hyperlinks ausgelöst. Innerhalb dieser Funktion wird zunächst ein neues Datumsobjekt namens `neuesDatum` erzeugt. Dem Objekt wird eine Datumsangabe zugewiesen. Das hier verwendete Datum wird mittels der **`alert()`**-Methode ausgegeben. Im nächsten Schritt wird der Monatstag geändert. Hierzu wird die Angabe `neuesDatum`**`.setDate()`** notiert. Als Wert wird hier die `23` verwendet. Zur Kontrolle wird abschließend der veränderte Monatstag in einem Meldungsfenster angezeigt.

11.14 Volles Jahr setzen

Über die **setFullYear()**-Eigenschaft kann die Jahreszahl, welche in einem Datumsobjekt gespeichert wurde, verändert werden. Als Parameter wird hier die gewünschte Jahreszahl erwartet. Die nachfolgende Syntax bewirkt, dass die bestehende Jahreszahl `2001` auf `2002` verändert wird. Das Ergebnis wird als Beschriftungstext auf einem Formular-Button angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function vollesJahr()
{
neuesDatum = new Date("2002","08","10","1","12","10","34");
neuesDatum.setFullYear( "2003" );
document.Formular.Ausgabe.value = neuesDatum.getFullYear();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="button" value="Hierher" name="Ausgabe">
<BR>
<INPUT type="button" value="Anzeigen" onclick="vollesJahr()">
</FORM>
</BODY>
</HTML>
```

Listing 11.13: Das neue Jahr wird auf dem Button angezeigt.

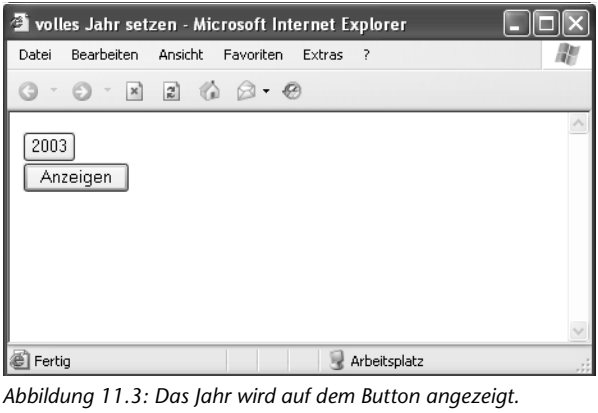


Abbildung 11.3: Das Jahr wird auf dem Button angezeigt.

Ausgelöst wird die Funktion **vollesJahr()** durch Anklicken des Formular-Buttons. Innerhalb der Funktion wird zunächst ein neues Datumsobjekt mit einer Datumsangabe namens neuesDatum erzeugt. Über die Angabe neuesDatum**.setFullYear()** wird auf das Datumsobjekt zugegriffen und die Jahreszahl durch den Parameter 2002 verändert. Abschließend wird der Wert des Formular-Buttons mit dem aus neuesDatum**.getFullYear()** ausgelesenen Wert gleichgesetzt.

11.15 Stunden setzen

Werte zwischen ***0*** *und* ***23***

Mittels der **setHours()**-Methode kann die Stunde der Uhrzeit, welche innerhalb eines Datumsobjekts gespeichert wurde, verändert werden. Als Parameter wird ein Stunden-Wert erwartet. Hierbei ist darauf zu achten, dass ein Wert zwischen **0** und **23** übergeben wird. Die nachstehende Syntax bewirkt, dass innerhalb eines Dokuments, sowohl die aktuellen Stunden der Uhrzeit wie auch ein manipulierter Stunden-Wert von 11 angezeigt werden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
neuesDatum = new Date();
document.write( "Stunden: " + neuesDatum.getHours() );
neuesDatum.setHours( "11" );
document.write( " Stunden: " + neuesDatum.getHours() );
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.14: Die alte und die neue Stunde werden in das Dokument geschrieben.

Direkt mit dem Einlesen der Datei wird der Script-Bereich ausgeführt. Hierin wird zunächst das neue Datumsobjekt aktuell erzeugt. Dieses wird innerhalb der ersten **write()**-Anweisung dazu verwendet, die aktuelle Uhrzeit auszugeben. Im nächsten Schritt wird über neuesDatum**.setHours()** auf die Stundenangabe zugegriffen und dieser der Wert 11 zugewiesen. Auch dieser Wert wird mittels der **write()**-Methode dynamisch in das Dokument geschrieben.

11.16 Millisekunden setzen

Werte zwischen 0 und 999

Durch den Einsatz der **setMilliseconds()**-Methode können die Millisekunden eines Datumsobjekts geändert werden. Als Parameter wird hierbei die Anzahl der Millisekunden erwartet. Der vergebene Wert muss zwischen **0** und **999** liegen. In dem folgenden Beispiel werden die Millisekunden auf den Wert `10` gesetzt und dieser wird anschließend innerhalb eines einzeiligen Eingabefeldes angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Millisekunden() {
 neuesDatum = new Date();
 neuesDatum.setMilliseconds(10);
 document.Formular.Ausgabe.value = neuesDatum.getMilliseconds();
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Ausgabe">
</FORM>
<A href="javascript: Millisekunden()">zeige Millisekunden</A></p>
</BODY>
</HTML>
```

Listing 11.15: Der gesetzte Millisekundenwert wird in dem Eingabefeld angezeigt.

Die Funktion `Millisekunden()` wird durch Anklicken des Verweises ausgelöst. In der Funktion wird ein neues Datumsobjekt mit dem Namen `neuesDatum` erzeugt. Über `neuesDatum.setMilliseconds(10)` wird der Millisekundenwert in `10` geändert. Abschließend wird dem Wert des Eingabefeldes die Anweisung `neuesDatum.getMilliseconds()` zugewiesen. Dies bewirkt, dass innerhalb des Eingabefeldes der modifizierte Millisekunden-Wert angezeigt wird.

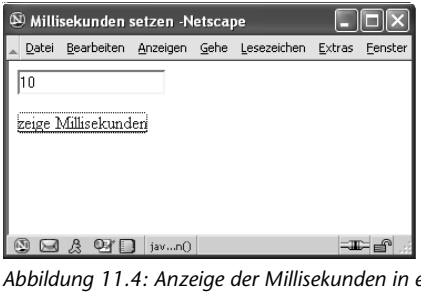


Abbildung 11.4: Anzeige der Millisekunden in einem Eingabefeld

11.17 Minuten setzen

Werte zwischen ***0*** *und* ***59***

Über die **setMinutes()**-Methode können die Minuten eines Datumsobjekts verändert werden. Als Parameter wird der neue Minutenwert erwartet. Dieser muss zwischen **0** und **59** liegen. Im folgenden Beispiel werden die Minuten der aktuellen Uhrzeit sowie der neu gesetzte Minuten-Wert 40 dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function setzeMinuten()
{
neuesDatum = new Date();
document.write( "alte Minuten: " + neuesDatum.getMinutes() );
neuesDatum.setMinutes(40);
document.write( "neue Minuten: " + neuesDatum.getMinutes() );
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:setzeMinuten()">Minuten</A>
</BODY>
</HTML>
```

Listing 11.16: Die alten und die neuen Minuten werden in das Dokument geschrieben.

Durch Anklicken des Verweises wird die Funktion `setzeMinuten()` aufgerufen. In der Funktion wird das neue Datumsobjekt `neuesDatum` erzeugt. Über die **write()**-Methode, in welcher ein beschreibender Text sowie die Anweisung `neuesDatum.getMinutes()` notiert sind, werden die aktuellen Minuten ermittelt und in das Dokument geschrieben. Im nächsten Schritt wird dem Datumsobjekt über `neuesDatum.setMinutes(40)` der neue Minutenwert zugewiesen. Auch dieser wird anschließend mittels der **write()**-Methode ausgegeben.

11.18 Monat setzen

Die **setMonth()**-Methode ermöglicht das Verändern des innerhalb eines Datumsobjekts gespeicherten Monatwerts. Erwartet wird als Parameter der neue Monat in Form eines numerischen Wertes. Durch die nachstehende Syntax werden der aktuelle Monat sowie ein neuer, mit dem zugewiesenen Wert *7*, dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function setzeMonat()
{
neuesDatum = new Date( "2002", "11", "10");
document.write( "alter Monat: " + neuesDatum.getMonth() );
neuesDatum.setMonth(7);
document.write( "  neuer Monat: " + neuesDatum.getMonth() );
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:setzeMonat()">Zeige neuen Monat</A>
</BODY>
</HTML>
```

Listing 11.17: Der alte und der neue Monat werden in das Dokument geschrieben.

Ausgelöst wird die Funktion `setzeMonat()` durch Anklicken des Hyperlinks. In der Funktion wird das neue Datumsobjekt `neuesDatum` erzeugt. Diesem wird das Datum `10.11.2002` zugewiesen. Über die **`write()`**-Methode wird der Monatswert `11` ausgegeben. Im nächsten Schritt wird über die Anweisung `neuesDatum`**`.setMonth(7)`** der Monat auf den Wert *7* gesetzt. Ausgelesen wird dieser Wert anschließend über die Angabe `neuesDatum`**`.getMonth()`** und durch die **`write()`**-Methode in das Dokument geschrieben.

11.19 Sekunden setzen

Werte zwischen 0 und 59

Über die **`setSeconds()`**-Methode kann der Sekundenwert der im Datumsobjekt gespeicherten Uhrzeit verändert werden. Als Parameter wird eine Sekundenanweisung, deren Wert zwischen **`0`** und **`59`** liegt, erwartet. Im folgenden Beispiel wird ein Datumsobjekt erzeugt und dessen Sekundenwert von `11` in `58` geändert.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function AnzahlSekunden()
{
neuesDatum = new Date( "98", "01", "11", "8","30", "11", "35" );
neuesDatum.setSeconds(58);
alert( "Sekunden: " + neuesDatum.getSeconds() );
```

```
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:AnzahlSekunden()">Zeige die Sekunden</A>
</BODY>
</HTML>
```

Listing 11.18: Die gesetzten Sekunden werden in einem Meldungsfenster ausgegeben.

Die `AnzahlSekunden()`-Funktion wird durch Anklicken des Verweises ausgelöst. Innerhalb der Funktion wird zunächst das neue Datumsobjekt `neuesDatum` erzeugt. Diesem wird eine vollständige Datumsangabe inklusive eines Sekundenwerts von `11` zugewiesen. Durch die Anweisung `neuesDatum.setSeconds(58)` wird der Sekundenwert des Datumobjekts in `58` geändert. Der neue Sekundenwert wird innerhalb der **`alert()`**-Methode durch die Angabe `neuesDatum.getSeconds` ermittelt und ausgegeben.

11.20 Datum und Uhrzeit setzen

Durch den Einsatz der **`setTime()`**-Methode kann der gesamte Inhalt des Datumobjekts verändert werden. Als Parameter wird ein numerischer Wert erwartet. Dieser wird von der **`setTime()`**-Methode als die Anzahl der seit dem 1.1.1970 um 0:00:00 Uhr vergangenen Millisekunden interpretiert. Um einen geeigneten Wert, der hier eine sinnvolle Datumsangabe nach sich zieht, zu ermitteln, kann die **`getTime()`**-Methode zu Hilfe genommen werden. Die nachstehende Syntax setzt den gesamten Inhalt des Datumsobjekts neu und schreibt diesen dynamisch in das Dokument.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function DatumUhrzeit()
{
neuesDatum = new Date( );
neuesDatum.setTime( "546566546547" );
document.write( "Jahr: " + neuesDatum.getFullYear());
document.write( "Monat: " + neuesDatum.getMonth() );
document.write( "Tag: " + neuesDatum.getDate() );
document.write( "Stunden: " + neuesDatum.getHours() );
document.write( "Minuten: " + neuesDatum.getMinutes() );
}
//-->
```

```
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:DatumUhrzeit()">setze Zeit</A>
</BODY>
</HTML>
```

Listing 11.19: Das neu definierte Datum wird in das Dokument geschrieben.

Innerhalb der `DatumUhrzeit()`-Funktion, welche durch Anklicken des Hyperlinks aufgerufen wird, wird das Datumsobjekt `neuesDatum` neu erzeugt. Durch die **`setTime()`**-Methode wird das Datum neu gesetzt. Hierzu wird der Wert `546566546547` verwendet. Das hieraus ermittelte Datum, der `28.3.1987`, wird anschließend über die **`write()`**-Methode in das Dokument geschrieben. Hierzu wird über die entsprechenden Methoden auf das zurückgelieferte Datum zugegriffen.

11.21 Die Jahreszahl ändern

Über die **`setYear()`**-Methode kann die in einem Datumobjekt gespeicherte Zahl geändert werden. Als Parameter wird das neue Jahr erwartet. So kann beispielsweise durch die Anweisung **`setYear(`**`2000`**`)`** das Jahr des Datumsobjekts in `2000` geändert werden. Im folgenden Beispiel wird das aktuelle Jahr ermittelt und in einem Meldungsfenster ausgegeben. Anschließend werden hiervon `10` Jahre abgezogen und das neue Jahr wird ebenfalls in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var neuesDatum = new Date();
var Jahr = neuesDatum.getYear();
alert(Jahr);
neuesDatum.setYear(Jahr-10);
Jahr = neuesDatum.getYear();
alert(Jahr);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 11.20: Die neue und alte Jahreszahl werden ausgegeben.

Zunächst wird ein neues Datumsobjekt erzeugt und dieses in der Variablen `neuesDatum` gespeichert. In der `Jahr`-Variablen wird über die **`getYear()`**-Methode das Jahr des Datumsobjekts ermittelt und über die **`alert()`**-Methode ausgegeben. Im nächsten Schritt wird mittels der **`setYear()`**-Methode das Jahr neu gesetzt. Hierzu wird das aktuelle Jahr mit dem Wert `10` subtrahiert. Das neu gesetzte Jahr wird ebenfalls durch die **`alert()`**-Methode ausgegeben.

11.22 Zeit ins GMT-Format umwandeln

IETF-Standard

Nicht immer ist die Ausgabe von Datumsangaben ideal. Durch die **`toGMTString()`**-Methode kann ein übergebenes Datumsobjekt nach dem IETF-Standard konvertiert werden. Wurde also beispielsweise das Datum *`16.12.2002 13:51:00`* gespeichert, liefert die **`toGMTString()`**-Methode den Wert *`Mon, 16 Dec 2002 12:51:00`* zurück. Beachten Sie, dass diese Angaben jeweils zeitzonenabhängig sind. Befindet sich der Anwender in einer anderen Zeitzone, wird dies bei der Konvertierung berücksichtigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
 var neuesDatum = new Date();
 alert(neuesDatum.toGMTString());
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 11.21: Das umgewandelte Datum wird in einem Meldungsfenster angezeigt.

In dem Beispiel wird ein neues Datumsobjekt erzeugt. Dieses wird in der Variablen `neuesDatum` gespeichert. Anschließend erfolgt durch die **`toGMTString()`**-Methode die Konvertierung nach dem IETF-Standard. Die Ausgabe des konvertierten Datums wird durch die **`alert()`**-Methode umgesetzt.

11.23 Zeit in lokales Format konvertieren

Die **`toLocaleString()`**-Methode wandelt die in einem Datumobjekt gespeicherten Werte in eine Zeichenkette um. Beachten Sie, dass diese Zeichenkette je nach verwendetem Browser variiert. Im nachstehenden Beispiel wird das übergebene Datum 2002,01,11 in die Zeichenkette „Montag, 11. Februar 2002 00:00:00" umgewandelt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
neuesDatum = new Date( "2002", "01", "11");
document.write( neuesDatum.toLocaleString() );
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 11.22: Das neue Datum wird dynamisch in das Dokument geschrieben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgeführt. Es wird ein neues Datumsobjekt namens `neuesDatum` erzeugt. Als Wert wird dieser der `11.1.2002` übergeben. Durch die Angabe `neuesDatum.toLocal-String()` wird dieser Wert umgewandelt und mittels der **`write()`**-Methode dynamisch in das Dokument geschrieben.

11.24 GMT-Zeit seit dem 1.1.1970 ermitteln

Die **`UTC()`**-Methode liefert die Millisekunden zurück, die zwischen dem 1.1.1970 und einem übergebenen Wert vergangen sind. Es werden in jedem Fall drei Parameter, nämlich diejenigen, die das Datum bestimmen, benötigt. Zusätzlich ist die Notation dreier Parameter möglich, welche die Uhrzeit spezifizieren. Folgende Parameter müssen bzw. können verwendet werden.

mögliche Parameter

- 1 – Jahreszahl
- 2 – Monat
- 3 – Wochentag
- 4 – Stunden (optional)
- 5 – Minuten (optional)
- 6 – Sekunden (optional)

Beachten Sie, dass sowohl die Datumsangaben wie auch die Uhrzeit in numerischer Form übergeben werden müssen. Die nachstehende Syntax gibt die Anzahl der zwischen dem `1.1.1970` und dem `7.7.2002` um `22.14.05` vergangenen Millisekunden in einem Meldungsfenster aus.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
```

```
function zeigeWert()
{
 alert(Date.UTC(2002,7,07,22,14,05));
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<P onmouseover="zeigeWert()">Wie die Zeit vergeht</P>
</BODY>
</HTML>
```

Listing 11.23: Die vergangenen Millisekunden werden in einem Meldungsfenster angezeigt.

Um die Funktion `zeigeWert()` auszulösen, wird der Event-Handler **`onmouseover`** verwendet. In der Funktion werden der **`UTC()`**-Methode das Datum sowie die Uhrzeit übergeben. Der Rückgabewert wird mittels der **`alert()`**-Methode ausgegeben.

11.25 Wie lange ist der Nutzer schon auf der Seite?

ein guter Zusatzservice

Wer kennt es nicht? Auf einer Seite wird die Zeit angezeigt, die man auf dieser bislang verbracht hat. Ob dies in jedem Fall eine sinnvolle Anwendung ist, ist allerdings fraglich. Schließlich möchte man ja im Normalfall den Anwender so lange wie möglich an seine Seite binden. Dennoch kann eine solch interne Uhr ein guter Zusatzservice für den Anwender sein. In unserem Beispiel wird dem Anwender die bislang vergangene Zeit in einem einzeiligen Eingabefeld angezeigt. Nach dem Laden der Seite ergibt sich ein Anblick, wie er in Abbildung 11.5 zu sehen ist.

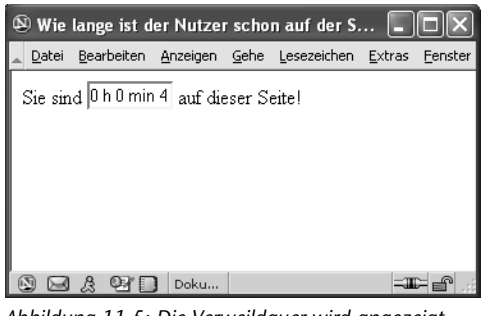


Abbildung 11.5: Die Verweildauer wird angezeigt.

Es gibt, wie immer, verschiedene Möglichkeiten, unser Problem zu lösen. In unserem Beispiel wird zunächst das aktuelle Datum ermittelt. Im nächsten Schritt benötigen wir einen Timer, durch den die Funktion

nach einer Sekunde aufgerufen wird. Es gäbe zur Realisierung dieser Aufgabe noch einen anderen Weg. Dieser würde ohne die Ermittlung der aktuellen Uhrzeit arbeiten. Problematisch hierbei ist allerdings, dass diese Uhr etwas nachgehen würde. Korrekter ist also der hier vorgestellte Weg, auch wenn dieser etwas umständlicher ist.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
   var  neuesDatum = new Date();
   function vergangeneZeit()
     {
     Datum = new Date();
     Zeit=(Datum.getTime() - neuesDatum.getTime())/1000;
     Stunden=Math.floor(Zeit/3600);
     Minuten=Math.floor((Zeit-3600*Stunden)/60);
     Sekunden=Math.round(Zeit-3600*Stunden-60*Minuten);
window.document.Formular.zeitbox.value = Stunden + " h " + Minuten +" min " +
Sekunden + " s";
     window.setTimeout('vergangeneZeit()',1000);
     }
   window.setTimeout('vergangeneZeit()',1000);
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
Sie sind <input size=9 name="zeitbox"> auf dieser Seite!
</FORM>
</BODY>
</HTML>
```

Innerhalb der Variablen `Zeit` wird das Eingangsdatum vom neu ermittelten Datum subtrahiert und dieser Wert mit `1000` multipliziert. Die nächsten Variablendeklarationen dienen der Berechnung der Stunden, Minuten und Sekunden. Wobei die **`floor()`**-Methode die nächstniedrigere Ganzzahl ermittelt. Die **`round()`**-Methode wiederum ermittelt die kaufmännisch gerundete Zahl. Die Werte der drei Variablen werden dem einzeiligen Eingabefeld als Wert zugewiesen. Um eine dynamische Uhranzeige zu realisieren, muss die Funktion `vergangeneZeit()` kontinuierlich jede Sekunde aufgerufen werden. Hierzu wird die **`setTimeout()`**-Methode eingesetzt.

11.25.1 Das aktuelle Datum ausgeben

Datum in deutscher Sprache

Zwar lässt sich das Datum mittels der in diesem Kapitel vorgestellten Methoden ausgeben, für komfortable Anwendungen sollten aber zu-

sätzliche Modifikationen vorgenommen werden. So ist es beispielsweise für den deutschen Sprachraum nicht ideal, dass Datumsangaben in englischer Sprache ausgegeben werden. In unserem Beispiel wollen wir uns aus diesem Grund darauf konzentrieren, wie sich eine Datumsausgabe in deutscher Sprache realisieren lässt. Schauen wir uns zunächst das Ergebnis dieser Aufgabe in der Abbildung 11.6 an.

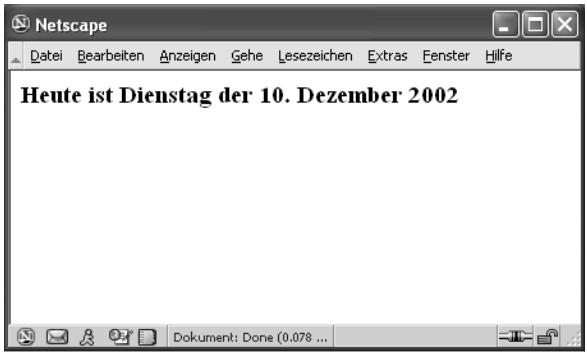


Abbildung 11.6: Eine formatierte Datumsausgabe

Wie Sie sehen, werden der Wochentag und der Monatsname in deutscher Sprache angezeigt. Zusätzlich wird das Datum in einen beschreibenden Text eingebettet. Um eine solche Datumsausgabe umzusetzen, müssen zwei Arrays angelegt werden. Im ersten Array werden die Wochentage und im zweiten die Monatsnamen abgelegt. Natürlich ist es auch hier so, dass zunächst das aktuelle Datum so ermittelt werden muss, wie Sie dies in diesem Kapitel bisher kennen gelernt haben. Die Umwandlung in das gewünschte Ausgabeformat ist dann dank der Verwendung der beiden Arrays recht simpel.

```
<HTML>
<HEAD>
<SCRIPT type=text/javascript>
<!--
  Wochentag = new Array();
  Wochentag[0] = "Sonntag";
  Wochentag[1] = "Montag";
  Wochentag[2] = "Dienstag";
  Wochentag[3] = "Mittwoch";
  Wochentag[4] = "Donnerstag";
  Wochentag[5] = "Freitag";
  Wochentag[6] = "Sonnabend";
  Monat = new Array();
  Monat[0] = "Januar";
  Monat[1] = "Februar";
  Monat[2] = "März";
```

```
  Monat[3] = "April";
  Monat[4] = "Mai";
  Monat[5] = "Juni";
  Monat[6] = "Juli";
  Monat[7] = "August";
  Monat[8] = "September";
  Monat[9] = "Oktober";
  Monat[10] = "November";
  Monat[11] = "Dezember";
  neuesDatum = new Date();
  aktuellesJahr  = neuesDatum.getYear();
aktuellesJahr=(aktuellesJahr<2000)?1900+aktuellesJahr:
aktuellesJahr;
aktuellerMonat = neuesDatum.getMonth();
  setzeTag = neuesDatum.getDay();
  aktuellerTag   = neuesDatum.getDate();
  generiereDatum  = Wochentag[setzeTag] + ' der ';
  generiereDatum += aktuellerTag + '. ';
  generiereDatum += Monat[aktuellerMonat] + ' ';
  generiereDatum += aktuellesJahr;
//-->
</SCRIPT>
</HEAD>
<BODY>
<H3>Heute ist
<script type=text/javascript>
document.write(generiereDatum);
</SCRIPT>
</H3>
</BODY>
</HTML>
```

Zunächst werden die beiden Arrays `Wochentag` und `Monat` erzeugt. Hierin werden dann die Wochentage und Monatsnamen notiert. Beachten Sie, dass bei beiden Arrays die interne Zählung bei 0 beginnen muss. Auf Grund der Vorkenntnisse bezüglich Arrays und des **`date`**-Objekts ist die Syntax recht einfach zu verstehen. Aus diesem Grund schauen wir uns das Prinzip des Beispiels anhand der Ausgabe des Wochentags an. Innerhalb der Variablendeklaration `setzeTag` wird mittels der **`getDay()`**-Methode der aktuelle Wochentag ermittelt. Wie Sie bereits wissen, liefert die **`getDay()`**-Methode einen numerischen Wert zwischen **`0`** (für Sonntag) und **`6`** (für Samstag) zurück. Der hier gespeicherte Wert wird als Index für die Ausgabe des korrekten Elements aus dem Array `Wochentag[]` verwendet. Mit diesem Prinzip wird auch der Monatsname ermittelt. Gespeichert wird das ermittelte Datum in der Variablen `generiereDatum`. Die Ausgabe erfolgt mittels der **`write()`**-Methode.

11.26 Fragen und Übungen

1. Schreiben Sie ein JavaScript-Programm, welches die noch verbleibenden Sekunden bis zum Jahr 2005 in einem Meldungsfenster ausgibt.

2. Schreiben Sie ein Programm, welches den Anwender tageszeitabhängig begrüßt. Der Begrüßungstext soll in das Dokument geschrieben werden. Die Einteilung der Begrüßungen soll in vier Intervallen erfolgen. Überlegen Sie sich vor der Realisierung des Programms günstige Zeitintervalle.

12 Rechnen mit JavaScript

lernen

Ziele dieses Kapitels

Wie Sie mit JavaScript rechnen können, lernen Sie in diesem Kapitel. Wir beschäftigen uns hierzu mit den beiden Objekten **math** und **number**, wobei das **math**-Objekt dem Zweck von Berechnungen dient. Durch das **number**-Objekt erhalten Sie Zugriff auf die Eigenschaften numerischer Werte. Am Ende dieses Kapitels werden Sie u.a. in der Lage sein, vollständige Berechnungen numerischer Werte durchzuführen, Zahlen auf- und abzurunden sowie die Gültigkeit von Zahlen zu ermitteln.

12.1 Zugriff auf das Math-Objekt

keine Typzuweisung

Für Berechnungen in JavaScript wird das **Math**-Objekt verwendet. Hiermit können Sie beispielsweise Tangens- und Kosinusfunktionen realisieren. Im Gegensatz zu anderen Programmiersprachen, wie beispielsweise Pascal, müssen und können Sie in JavaScript den numerischen Werten keinen Typ zuweisen. Während in Pascal die beiden Zahlen 9 und 9.5 Werte von unterschiedlichem Typ sind, nämlich integer und real, muss dies in JavaScript nicht angegeben werden. In JavaScript findet eine automatische Typkonvertierung statt. Das erleichtert zwar fürs Erste das Programmieren, hat mit einer korrekten Programmierung allerdings wenig zu tun. In der JavaScript-Version 2.0 dürfte dieser Makel dann allerdings beseitigt werden. Bis dies dann allerdings von den WWW-Browsern interpretiert werden kann, wird wohl noch einige Zeit ins Land gehen. Der Zugriff auf das **Math**-Objekt und dessen Eigenschaften und Methoden wird folgendermaßen realisiert:

```
Math.Eigenschaft
Math.Methode()
```

Hinter dem Schlüsselwort **Math** wird die gewünschte Eigenschaft oder Methode notiert. Welche Eigenschaften und Methoden auf dieses Objekt anwendbar sind, erfahren Sie im Laufe dieses Abschnitts. Exemplarisch soll die folgende Syntax zwei typische Notationsformen demonstrieren.

```
Math.LN2
Math.random()
```

Bezüglich der Nachkommastellen unterliegen Programmiersprachen, und eben auch JavaScript, Konventionen. Diese sind allerdings systemabhängig. Für moderne Betriebssysteme sei hier ein Richtwert von 32 Bit gegeben.

12.1.1 Eulersche Konstante

E* ist circa *2.178

E liefert die Eulersche Konstante. Sicher ist, dass es sich bei **E** um eine der wichtigsten Zahlen der Mathematik handelt. Deren ungefährer Wert, von mir gerundet, liegt bei *2.178*. Diese Zahl wurde von Euler ausgiebig untersucht und trägt aus diesem Grund dessen **E** als Bezeichner.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
document.write("Konstante E: " + Math.E + "<BR>");
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 12.1: Die Eulersche Konstante wird ausgegeben

In dem gezeigten Beispiel wird die Konstante **E** über die Anweisung **document.write()** dynamisch in das Dokument geschrieben. Als Ausgabe erscheint im Browserfenster der Wert *2.718281828459045*.

12.1.2 Natürlicher Logarithmus von 2

LN2* ist circa *0,6932

Die **LN2**-Eigenschaft speichert den natürlichen Logarithmus von 2. Der ermittelte Wert beträgt hierbei circa *0,6932*. Anhand der nachstehenden Syntax wird dieser Wert in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
alert(Math.LN2);
}
```

```
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeWert()">nat&uuml;rlicher Logarithmus von zwei</A>
</BODY>
</HTML>
```

Listing 12.2: Der natürliche Logarithmus von 2 wird in einem Meldungsfenster angezeigt.

Das Anklicken des Hyperlinks ruft die Funktion `zeigeWert()` auf. In dieser Funktion wird der natürliche Logarithmus von 2 durch die Anweisung **`Math.LN2`** ermittelt und anhand der **`alert()`**-Methode ausgegeben.

12.1.3 Natürlicher Logarithmus von 10

Über die **`LN10`**-Eigenschaft kann der natürliche Logarithmus von 10 ermittelt werden. Als Wert wird hierbei circa *2,3026* ermittelt. Anhand der nachstehenden Syntax wird der natürliche Logarithmus von 10 ermittelt und dynamisch in das Dokument geschrieben.

LN10 ist circa 2,3026

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
document.write(Math.LN10);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeWert()">nat&uuml;rlicher Logarithmus von zehn</A>
</BODY>
</HTML>
```

Listing 12.3: Der natürliche Logarithmus von 10 wird in das Dokument geschrieben.

Aufgerufen wird die Funktion `zeigeWert()` durch Anklicken des Verweises. Über **`Math.LN10`** wird der natürliche Logarithmus von 10 ermittelt und mittels der **`write()`**-Methode ausgegeben.

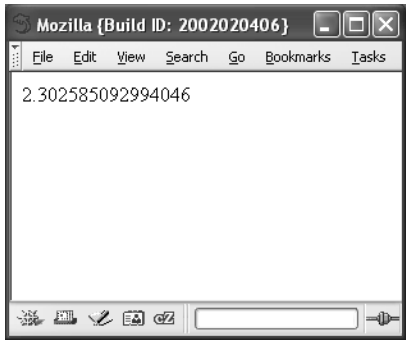


Abbildung 12.1: Der natürliche Logarithmus von 10.

12.1.4 Konstanter Logarithmus von 2

LOG2E ist circa 1,443

Die **LOG2E**-Eigenschaft ermittelt den Logarithmus der Eulerschen Konstante zur Basis 2. Dieser Wert liegt bei circa *1,443*. Das folgende Beispiel ermittelt den genannten Wert und gibt diesen in einem Meldungsfenster aus.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
alert(Math.LOG2E);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeWert()">Logarithmus von e</A>
</BODY>
</HTML>
```

Listing 12.4: Der konstante Logarithmus von 2 wird ausgegeben.

Die Funktion `zeigeWert()` wird durch Anklicken des Verweises ausgelöst. Innerhalb der Funktion wird über die Anweisung **`Math.LOG2E`** der Logarithmus von e zur Basis zwei ermittelt. Der zurückgelieferte Wert wird mittels der **`alert()`**-Methode in einem Meldungsfenster angezeigt.

12.1.5 Konstanter Logarithmus von 10

LOG10E ist circa 0,434

Die **LOG10E**-Eigenschaft ermittelt den Logarithmus der Eulerschen Konstante zur Basis 10. Ermittelt wird hierbei ein Wert von circa **0,434**. Durch die nachstehende Syntax wird dieser Wert dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
document.write("Konstante LOG10E: " + Math.LOG10E);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 12.5: Der konstante Logarithmus von 10 wird in das Dokument geschrieben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgelöst. Durch die Anweisung **Math.LOGE10** wird der Logarithmus von e zur Basis 10 ermittelt. Der zurückgelieferte Wert wird einschließlich eines beschreibenden Textes durch die **write()**-Methode ausgegeben.

12.1.6 PI

PI ist circa 3,14159

Mit der Eigenschaft **PI** wird die Kreiszahl PI mit einem Wert von circa 3,14159 ermittelt. Die nachstehende Syntax bewirkt, dass dieser Wert dynamisch in das Dokument geschrieben wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
document.write(Math.PI);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeWert()">PI</A>
</BODY>
</HTML>
```

Listing 12.6: Der Wert von PI wird angezeigt.

Der Script-Bereich wird mit dem Einlesen der Datei ausgelöst. Anhand der Anweisung **Math.PI** wird der Wert von PI ermittelt. Dessen Ausgabe wird hier mittels der **write()**-Methode realisiert.

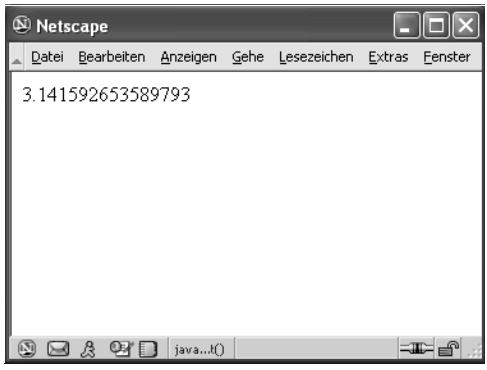


Abbildung 12.2: Der Wert von PI wird in das Dokument geschrieben.

12.1.7 Konstante Quadratwurzel aus 0,5

SQRT1_2 ist circa 0,707

Die **SQRT1_2**-Eigenschaft berechnet die Quadratwurzel von `0,5`. Der hierbei ermittelte Wert liegt bei circa `0,707`. Die nachstehende Syntax bewirkt, dass die Quadratwurzel von `0,5` in einem Meldungsfenster angezeigt wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeWert()
{
alert(Math.SQRT1_2);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:zeigeWert()">konstante Quadratwurzel aus
0,5</A>
</BODY>
</HTML>
```

Listing 12.7: Die Ausgabe der konstanten Quadratwurzel von `0,5`

Die Funktion `zeigeWert()` wird durch Anklicken des Hyperlinks ausgelöst. Mittels der **Math.SQRT1_2**-Anweisung wird die Quadratwurzel von `0,5` ermittelt. Der hierbei gespeicherte Wert wird über die **alert()**-Methode ausgegeben.

12.1.8 Konstante Quadratwurzel aus 2

SQRT2 ist circa 1,414

Die **SQRT2**-Eigenschaft ermittelt die Quadratwurzel von 2. Der gespeicherte Wert beträgt hierbei circa **1,414**. Die folgende Syntax bewirkt, dass dieser Wert dynamisch in das Dokument geschrieben wird.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
document.write("Konstante SQRT2: " + Math.SQRT2 );
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 12.8: Die konstante Quadratwurzel aus 2 wird in das Dokument geschrieben.

Der Script-Bereich wird direkt mit dem Einlesen der Datei ausgelöst. Hierin wird die Quadratwurzel von 2 durch die Anweisung **Math.SQRT2** gespeichert und mittels der **write()**-Methode ausgegeben.

12.1.9 Absolutbetrag einer Zahl

gilt auch bei negativen Zahlen

Die **Abs()**-Methode liefert den Absolutbetrag einer Zahl. Als Parameter wird eine Zahl erwartet. Es wird in jedem Fall eine positive Zahl gespeichert. Dies gilt auch dann, wenn eine negative Zahl übergeben wurde. Bei der Übergabe eines nicht numerischen Wertes wird **NaN** gespeichert. Im folgenden Beispiel befinden sich zwei einzeilige Eingabefelder. Das erste dient der Eingabe eines numerischen Wertes. In dem zweiten wird dessen Absolutbetrag angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Absolutbetrag()
{
Formular.Ausgabe.value = Math.abs(Formular.Eingabe.value);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
```

```
<INPUT type="text" name="Eingabe"><INPUT name="Ausgabe">
<INPUT type=button value="Denk Positiv" onclick="Absolutbetrag()">
</FORM>
</BODY>
</HTML>
```

Listing 12.9: Der Absolutbetrag der eingegebenen Zahl wird in dem Eingabefeld ausgegeben.

Der Event-Handler **onclick** löst die Funktion `Absolutbetrag()` aus. Hierin wird der Wert des zweiten Eingabefeldes `Ausgabe` mit dem Wert des ersten Eingabefeldes gleichgesetzt. Zusätzlich wird der Wert des ersten Eingabefeldes durch die Anweisung **Math.Abs()** in den Absolutbetrag umgewandelt.

12.1.10 Arcus Cosinus

die inverse Cosinus-Funktion

Die **acos()**-Methode erwartet eine Zahl als Parameter und liefert von dieser den Arcus Cosinus zurück. Der Arcus Cosinus ist die inverse Funktion des Cosinus liefert also dessen Kehrwert. Es wird als Parameter ein numerischer Wert zwischen -1 und 1 erwartet. Im folgenden Beispiel befinden sich zwei einzeilige Eingabefelder. Der Wert des ersten Eingabefeldes wird hierbei gespeichert und hiervon der Arcus Cosinus gebildet. Der ermittelte Wert wird im zweiten Eingabefeld angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function AcosCosinus()
{
Formular.Ausgabe.value = Math.acos(Formular.Eingabe.value);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe"><INPUT type="text" name="Ausgabe">
<INPUT type=button value="Arcus Cosinus " onclick=" AcosCosinus()">
</FORM>
</BODY>
</HTML>
```

Listing 12.10: Der Arcus Cosinus der eingegebenen Zahl wird in dem zweiten Eingabefeld ausgegeben.

Ausgelöst wird die Funktion **AcosCosinus()** durch Anklicken des Formular-Buttons. Innerhalb dieser Funktion wird der Wert des ersten mit dem des zweiten Eingabefeldes gleichgesetzt. Zusätzlich wird der Wert des ersten Eingabefeldes durch die Anweisung **Math.acos()** in den Arcus Kosinus umgewandelt. Der ermittelte Wert wird im zweiten Eingabefeld angezeigt. Wird kein numerischer Wert eingegeben oder liegt dieser außerhalb des Gültigkeitsbereichs, wird **NaN** angezeigt.

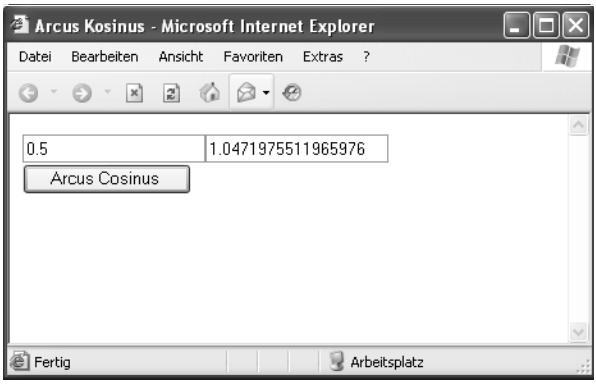


Abbildung 12.3: Der Arcus Cosinus wird ausgegeben.

12.1.11 Arcus Sinus

Achten Sie auf den Gültigkeitsbereich!

Die **asin()**-Methode liefert den Arcus Sinus einer Zahl. Als Parameter wird ein Zahlenwert erwartet. Beachten Sie, dass der übergebene Wert den Gültigkeitsbereich –1-1 nicht verlassen darf, da ansonsten **NaN** gespeichert wird. Im folgenden Beispiel befindet sich ein einzeiliges Eingabefeld. Von einem hierin notieren numerischen Wert wird der Arcus Sinus gebildet. Das Ergebnis wird anschließend in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function ArcusSinus()
{
alert(Math.asin(Formular.Eingabe.value));
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
```

```
<INPUT type=button value="Arcus Sinus" onclick="ArcusSinus()">
</FORM>
</BODY>
</HTML>
```

Listing 12.11: Der Arcus Sinus wird in einem Meldungsfenster angezeigt

Durch Anklicken des Formular-Buttons wird die `ArcusSinus()`-Funktion ausgelöst. Hierin wird der **`asin()`**-Methode als Zahl der Wert des Eingabefeldes `Eingabe` übergeben. Der ermittelte Wert wird durch die **`alert()`**-Methode ausgegeben.

12.1.12 Arcus Tangens

Die **`atan()`**-Methode ermittelt den Arcus Tangens einer übergebenen Zahl. Als Parameter wird ein numerischer Wert erwartet. Im folgenden Beispiel wird der Arcus Tangens eines innerhalb eines einzeiligen Eingabefeldes notierte Wertes ermittelt und in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function ArcusTangens()
{
alert(Math.atan(Formular.Eingabe.value));
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="button" value="Arcus Tangens" onclick="ArcusTangens()">
</FORM>
</BODY>
</HTML>
```

Listing 12.12: Der Arcus Tangens von 7 ist 1.4288992721907327

Der Event-Handler **`onclick`** löst die Funktion `ArcusTangens()` aus. Hierin wird auf den Wert des Eingabefeldes `Eingabe` zugegriffen. Dieser wird der **`atan()`**-Methode als Parameter übergeben. Das Ergebnis wird anschließend über die **`alert()`**-Methode ausgegeben.

12.1.13 Den Ankatheten-Winkel ermitteln

Es müssen zwei Zahlen angegeben werden.

Die **Athan2()**-Methode berechnet den Arcus Tangens. Als Parameter werden zwei numerische Werte, die durch Komma voneinander getrennt werden müssen, erwartet. Der erste Wert bezieht sich auf die Gegenkathete und der zweite auf die Ankathete. Als Ergebnis wird der an der Ankathete anliegende Winkel geliefert. Im folgenden Beispiel befinden sich zwei einzeilige Eingabefelder. Das erste erwartet den Wert der Gegenkathete und das zweite den der Ankathete. Der Winkel, der an der Ankathete anliegt, wird in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function ArcusTangens()
{
var Gegenkathete = document.Formular.GKT.value;
var Ankathete = document.Formular.AKT.value;
alert(Math.atan2(Gegenkathete,Ankathete));
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT name="GKT">
<INPUT name="AKT">
<INPUT type="button" value="Arcus Tangens" onclick=" ArcusTangens()">
</FORM>
</BODY>
</HTML>
```

Listing 12.13: Die Ausgabe des Ankatheten-Winkels in einem Meldungsfenster

Abbildung 12.4: Der Arcus Tangens wird ausgegeben.

Die Funktion `ArcusTangens()` wird durch Anklicken des Formular-Buttons ausgelöst. Innerhalb der Funktion werden die beiden Variablen `Gegenkathete` und `Ankathete` deklariert. Als Wert wird diesen der Inhalt der Felder `GKT` für die Variable `Gegenkathete` bzw. `AKT` für die Variable `Ankathete` zugewiesen. Über die Anweisung **`math.Atan2(`**`Gegenkathete,Ankathete`**`)`** wird der Winkel berechnet. Die **`alert()`**-Methode dient der Ausgabe des ermittelten Wertes.

12.1.14 Die nächsthöhere Zahl

Jede Zahl wird aufgerundet.

Die **`ceil()`**-Methode rundet eine Zahl auf die nächsthöhere Ganzzahl. Als Parameter wird ein Zahlenwert erwartet. Ist dieser bereits eine Ganzzahl, wird der Wert nicht aufgerundet. Bei der Übergabe des Wertes 2.2 wird auf 3 gerundet. Wurde beispielsweise der negative Wert -2.9 übergeben, liefert die **`ceil()`**-Methode den Wert –2. Im nachstehenden Beispiel stehen zwei einzeilige Eingabefelder zur Verfügung. Der im ersten Feld notierte Wert wird auf die nächsthöhere Ganzzahl aufgerundet und im zweiten Eingabefeld angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function naechsthoehereZahl()
{
Formular.Ausgabe.value = Math.ceil(Formular.Eingabe.value);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe"><INPUT type="text" name="Ausgabe">
<INPUT type="button" value="Sinus" onclick="naechsthoehereZahl()">
</FORM>
</BODY>
</HTML>
```

Listing 12.14: Die nächsthöhere Zahl wird im Eingabefeld angezeigt.

Ausgelöst wird die `naechsthoehereZahl()`-Funktion durch Anklicken des Formular-Buttons. Innerhalb der Funktion wird der Wert des ersten Eingabefeldes `Ausgabe` mit dem Wert des zweiten Eingabefeldes gleichgesetzt. Zusätzlich hierzu wird der Wert des ersten Eingabefeldes durch die Anweisung **`Math.ceil()`** auf die nächsthöhere Ganzzahl aufgerundet.

12.1.15 Nächstniedrigere ganze Zahl

Jede Zahl wird abgerundet.

Durch den Einsatz der **floor()**-Methode wird eine Zahl auf ihren nächstniedrigeren Ganzzahlanteil gerundet. Als Parameter wird ein numerischer Wert erwartet. Handelt es sich bei diesem um eine Ganzzahl, findet keine Rundung statt. Wird beispielsweise der Wert 4.9 übergeben, liefert **floor()** den Wert 4. Bei der Übergabe des negativen Wertes -4.9 wird 5 ermittelt. Die nachstehende Syntax dient dazu, den innerhalb eines Eingabefelds notierten Wert auf die nächste Ganzzahl abzurunden und das Ergebnis in einem Meldungsfenster anzuzeigen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function naechsniedrigereZahl()
{
alert(Math.floor(Formular.Eingabe.value));
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type=button value="Abrunden" onclick="naechsniedrigereZahl()">
</FORM>
</BODY>
</HTML>
```

Listing 12.15: Die Eingabe 5.9 liefert den Wert 5.

Die `naechsniedrigereZahl()`-Funktion wird durch den Event-Handler **onclick** ausgelöst. Innerhalb der Funktion wird der **floor()**-Methode als Parameter der Wert des einzeiligen Eingabefeldes `Eingabe` übergeben. Der ermittelte Wert wird anschließend über die **alert()**-Methode ausgegeben.

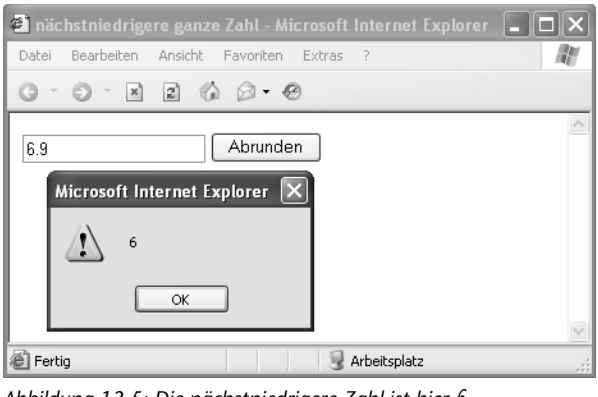


Abbildung 12.5: Die nächstniedrigere Zahl ist hier 6.

12.1.16 Cosinus

Die **cos()**-Methode liefert den Cosinus einer Zahl. Als Parameter wird ein numerischer Wert erwartet. Anhand der nachstehenden Syntax wird von einem innerhalb eines einzeiligen Eingabefeldes notieren Werts der Cosinus berechnet und das Ergebnis in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeKosinus()
{
alert(Math.cos(Formular.Eingabe.value));
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="button" value="Kosinus" onclick="zeigeKosinus()">
</FORM>
</BODY>
</HTML>
```

Listing 12.16: Der Cosinus von 3 ist -0.9899924966004454.

Durch den Event-Handler **onclick** wird die `zeigeKosinus()`-Funktion ausgelöst. In dieser Funktion wird der **cos()**-Methode der Wert des einzeiligen Eingabefeldes `Eingabe` als Parameter übergeben. Die Ausgabe des Cosinus-Werts erfolgt mittels der **alert()**-Methode.

12.1.17 Exponentialwert

Als Basis dient die Eulersche Konstante.

Die **exp()**-Methode liefert den Expotentialwert einer Zahl auf Basis der Eulerschen Konstanten. Als Parameter wird die Zahl erwartet. Die folgende Syntax dient dazu, den Exponentialwert eines innerhalb eines einzeiligen Eingabefeldes notieren Wertes zu ermitteln und diesen in einem zweiten Eingabefeld anzuzeigen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Exponentialwert()
{
```

```
Formular.Ausgabe.value = Math.exp(Formular.Eingabe.value);
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="text" name="Ausgabe">
<INPUT type=button value="Exponentialwert" onclick="Expotentialwert()">
</FORM>
</BODY>
</HTML>
```

Listing 12.17: Die Zahl 6 liefert den Exponentialwert 403.4287934927351.

Der Event-Handler **`onclick`** löst die Funktion `Exponentialwert()` aus. Hierin wird der Wert des Eingabefeldes `Ausgabe` mit dem des Feldes `Eingabe` gleichgesetzt. Der Wert des Eingabefeldes `Eingabe` wird der **`exp()`**-Methode als Parameter übergeben.

12.1.18 Natürlicher Logarithmus einer Zahl

Die **`log()`**-Methode liefert den natürlichen Logarithmus einer Zahl. Als Parameter wird ein numerischer Wert erwartet. Ist der übergebene Wert <=0, so wird der Wert -1,7977 (gerundet) zurückgegeben. Im folgenden Beispiel befinden sich zwei einzeilige Eingabefelder. Der natürliche Logarithmus des innerhalb des ersten Eingabefeldes notierten Zahlenwerts wird innerhalb des zweiten Feldes angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function Logarithmus()
{
Formular.Ausgabe.value = Math.log(Formular.Eingabe.value);
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="text" name="Ausgabe">
<INPUT type="button" value="Logarithmus" onclick="Logarithmus()">
</FORM>
</BODY>
</HTML>
```

Listing 12.18: Der natürliche Logarithmus von 2 ist 0.6931471805599453.

Die Funktion `Logarithmus()` wird durch den Event-Handler **`onClick`** ausgelöst. Innerhalb der Funktion wird der Wert des Eingabefeldes `Ausgabe` mit dem des Feldes `Eingabe` gleichgesetzt, wobei der Wert des Eingabefeldes `Eingabe` der **`log()`**-Methode als Parameter übergeben wird.

12.1.19 Die größere von zwei Zahlen ermitteln

Anhand der **`max()`**-Methode kann die größere von zwei Zahlen ermittelt werden. Als Parameter werden zwei numerische Werte erwartet. Im folgenden Beispiel befinden sich unter anderem drei Eingabefelder. Die ersten beiden dienen der Eingabe zweier Werte. Im dritten Feld wird die größere der beiden Zahlen angezeigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="groessereZahl">
<INPUT type="text" name="Wert1">
<INPUT type="text" name="Wert2">
<INPUT type="text" name="Ausgabe">
<INPUT type="button" value="gr&ouml;&szlig;te Zahl"
onclick="groessereZahl.Aus-
gabe.value=Math.max(groessereZahl.Wert1.value,groessereZahl.Wert2.value)">
</FORM>
</BODY>
</HTML>
```

Listing 12.19: Die größere Zahl wird im dritten Eingabefeld angezeigt.

Als Parameter werden der **`max()`**-Methode die Werte der beiden Eingabefelder `Wert1` und `Wert2` übergeben. Der hier ermittelte Wert wird dem Wert des `Ausgabe`-Feldes zugewiesen und somit dort angezeigt.

12.1.20 Die kleinere von zwei Zahlen ermitteln

Mittels der **`min()`**-Methode kann der kleinere von zwei Zahlenwerten ermittelt werden. Als Parameter werden zwei numerische Werte erwartet. Die nachstehende Syntax beinhaltet drei einzeilige Eingabefelder. Innerhalb der ersten beiden wird jeweils die Notation einer Zahl erwartet. Im dritten Feld wird die kleinere der beiden angezeigt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
```

```
<FORM name="kleinereZahl">
<INPUT type="text" name="Wert1">
<INPUT type="text" name="Wert2">
<INPUT type="text" name="Ausgabe">
<INPUT type="button" value="kleinste Zahl"
onclick="kleinereZahl.Ausgabe.value=Math.min(kleinereZahl.Wert1.value,
kleinereZahl.Wert2.value)">
</FORM>
</BODY>
</HTML>
```

Listing 12.20: Die kleinere Zahl wird im dritten Eingabefeld ausgegeben.

Als Parameter werden der **`min()`**-Methode die Werte der beiden Eingabefelder `Wert1` und `Wert2` zugewiesen. Der Rückgabewert wird dem Wert des Eingabefeldes `Ausgabe` zugewiesen.

Abbildung 12.6: Die kleinste Zahl ist hier 3.

12.1.21 Zahl hoch Exponent

Zwei Parameter werden erwartet.

Die **`pow()`**-Methode speichert den um einen angegebenen Exponenten potenzierten Wert. Als Parameter werden zwei numerische Werte erwartet. Wobei die erste Zahl (nennen wir sie x) die Basis und die zweite Zahl (nennen wir sie y) der Exponent ist. Die **`pow()`**-Methode berechnet hieraus die Potenz von y zur Basis x. Sollte es sich bei dem Ergebnis nicht um eine reelle Zahl handeln, wird der Wert **`0`** gespeichert. Im folgenden Beispiel befinden sich drei einzeilige Eingabefelder. Wobei sich der Wert des dritten Feldes aus der Potenz des zweiten zur Basis des ersten Feldes ergibt.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
```

```
<FORM name="Exponent">
<INPUT type="text" name="Wert1">
<INPUT type="text" name="Wert2">
<INPUT type="text" name="Ausgabe">
<INPUT type="button" value="zeige Exponent" onclick="Exponent.Aus-
gabe.value=Math.pow(Exponent.Wert1.value,Exponent.Wert2.value)">
</FORM>
</BODY>
</HTML>
```

Listing 12.21: 2 und 3 ergibt 8

Der **pow()**-Methode werden die Werte der beiden einzeiligen Eingabefelder `Wert1` und `Wert2` zugewiesen. Das Ergebnis wird im Feld `Ausgabe` angezeigt.

12.1.22 Zufallszahl zwischen 0 und 1

Mittels der **random()**-Methode kann eine Zufallszahl ermittelt werden. Diese liegt zwischen **0** und **1**. Diese Methode erwartet keinen Parameter. Im nachstehenden Beispiel wird eine Zufallszahl dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Zufallszahl()
{
document.write(Math.random());
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type=button value="Zufallszahl" onclick="Zufallszahl()">
</FORM>
</BODY>
</HTML>
```

Listing 12.22: Zufällige Zahlen können auch für zufällige Bilderwechsel eingesetzt werden.

Durch Anklicken des Formular-Buttons wird die Funktion `Zufallszahl()` ausgelöst. Innerhalb dieser Funktion wird über die Anweisung **Math.random()** eine Zufallszahl ermittelt. Diese wird über die **write()**-Methode ausgegeben.

12.1.23 Zahlen runden

kaufmännisches Runden

Die **round()**-Methode ermöglicht das Runden von numerischen Werten. Als Parameter wird eine Zahl erwartet. Für das Runden gelten die herkömmlichen mathematischen Regeln: Ist der Nachkommaanteil größer als 5, wird aufgerundet. Ist der Nachkommaanteil kleiner, wird abgerundet. Anhand der nachfolgend aufgeführten Syntax wird ein innerhalb des Eingabefeldes notierter numerischer Wert gerundet und das Ergebnis in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function rundeZahlen()
{
alert(Math.round(Formular.Eingabe.value));
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type=button value="gerundeter Wert" onclick="rundeZahlen()">
</FORM>
</BODY>
</HTML>
```

Listing 12.23: Die gerundete Zahl wird ausgegeben.

Die `rundeZahlen()`-Funktion wird durch den Event-Handler **onclick** aufgerufen. Als Parameter wird der **round()**-Methode der Wert des einzeiligen Eingabefeldes `Eingabe` übergeben. Der ermittelte Wert wird anschließend durch die **alert()**-Methode ausgegeben.

12.1.24 Sinus einer Zahl

Mittels der **sin()**-Methode kann der Sinus einer Zahl ermittelt werden. Als Parameter wird ein numerischer Wert erwartet. Innerhalb des nachfolgenden Beispiels befinden sich zwei einzeilige Eingabefelder. Der Sinus des innerhalb des ersten Feldes notierten Werts wird im zweiten Eingabefeld angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeSinus()
```

```
{
Formular.Ausgabe.value = Math.sin(Formular.Eingabe.value);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe"><INPUT type="text" name="Ausgabe">
<INPUT type="button" value="Sinus" onclick="zeigeSinus()">
</FORM>
</BODY>
</HTML>
```

Listing 12.24: Die Ausgabe des Sinus erfolgt im zweiten Eingabefeld.

Der Event-Handler **onclick** löst die Funktion `zeigeSinus()` aus. Innerhalb dieser Funktion wird der Wert des Eingabefeldes `Ausgabe` mit dem des Eingabefeldes `Eingabe` gleichgesetzt. Der Wert des Eingabefeldes `Eingabe` wird hierbei der **sin()**-Methode als Parameter übergeben.

12.1.25 Quadratwurzel

***NaN** bei negativen Zahlen*

Die **sqrt()**-Methode speichert die Quadratwurzel einer übergebenen Zahl. Als Parameter wird ein numerischer Wert erwartet. Bei der Eingabe eines negativen Zahlenwertes wird **NaN** gespeichert. Das folgende Beispiel hält ein einzeiliges Eingabefeld bereit. Die Quadratwurzel der Zahl, die innerhalb des Feldes notiert wird, wird dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Quadratwurzel()
{
var Feld = document.Formular.Eingabe.value;
document.write(Math.sqrt(Feld));
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type=button value="Quadratwurzel" onclick="Quadratwurzel()">
```

```
</FORM>
</BODY>
</HTML>
```

Listing 12.25: Die Quadratwurzel wird in das Dokument geschrieben.

Die `Quadratwurzel()`-Funktion wird durch Anklicken des Formular-Buttons ausgelöst. Innerhalb dieser Funktion wird der **`sqrt()`**-Methode der Wert des einzeiligen Eingabefeldes als Parameter übergeben. Die Ausgabe des Ergebnisses erfolgt mittels der **`write()`**-Methode.

12.1.26 Tangens einer Zahl

Die **`tan()`**-Methode ermittelt den Tangens einer übergebenen Zahl. Als Parameter wird ein numerischer Wert erwartet. Im folgenden Beispiel wird der Tangens des innerhalb eines einzeiligen Eingabefeldes notierten Wertes ermittelt und in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function zeigeTangens()
{
var Wert = document.Formular.Eingabe.value;
alert(Math.tan(Wert));
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="button" value="Tangens" onclick="zeigeTangens()">
</FORM>
</BODY>
</HTML>
```

Listing 12.26: Der Tangens einer eingegebenen Zahl wird ausgegeben.

Ausgelöst wird die Funktion `zeigeTangens()` durch Anklicken des Formular-Buttons. Der **`tan()`**-Methode wird als Parameter der Wert des Eingabefeldes `Eingabe` zugewiesen. Die Ausgabe des ermittelten Wertes erfolgt über die **`alert()`**-Methode.

12.2 Zugriff auf das Number-Objekt

Eigenschaften numerischer Werte

Um die Eigenschaften numerischer Werte zu ermitteln, wird das **Number**-Objekt verwendet. Während die eigentlichen Berechnungen über das **Math**-Objekt realisiert werden, können Sie durch das **Number**-Objekt eine Zahl auf ihre Gültigkeit hin überprüfen. Bezüglich der Zuweisung von Eigenschaften und Methoden bestehen beim **Number**-Objekt Unterschiede. Eigenschaften weisen Sie auf die folgende Weise zu:

```
Number.Eigenschaft
```

Hinter dem **Number**-Objekt wird durch einen Punkt getrennt die gewünschte Eigenschaft notiert. Als Beispiel sei hier die Verwendung der **Min_Value**-Eigenschaft gezeigt.

```
Number.Min_Value
```

Das **Number**-Objekt erwartet bezüglich der Methoden eine andere Notation. Hierbei muss zunächst ein numerischer Wert und dann durch einen Punkt getrennt die entsprechende Methode notiert werden.

```
Wert.Methode()
```

Bei diesem Wert kann es sich direkt um eine Zahl handeln. Ebenso ist aber auch die Verwendung einer Variablen möglich. Diesen Fall beschreibt das folgende kleine Beispiel. Hier wird der Variablen `x` der Wert `10.9999` zugewiesen. Dieser Wert wird durch die **toFixed()**-Methode mit nur noch zwei Nachkommastellen angezeigt.

```
x = 10.9999
x.toFixed(2)
```

Es besteht die Möglichkeit, ein neues **Number()**-Objekt zu erzeugen. Beachten Sie hierbei, dass dies häufig nicht sinnvoll ist. Schließlich lassen sich numerische Werte in JavaScript direkt bearbeiten.

```
new Number(Wert)
```

Hinter dem Schlüsselwort **new** wird das **Number()**-Objekt notiert. Innerhalb der Klammern kann nun eine Ziffer angegeben werden. Wollen Sie beispielsweise die Zahl *29* erzeugen, müsste **Number(***29***)** verwendet werden.

12.2.1 Die größte erlaubte Zahl

Über die **MAX_VALUE**-Eigenschaft kann die größte Zahl, die von JavaScript verarbeitet werden kann, ermittelt werden. Es handelt sich hierbei um den Zahlenwert *1.7976931348623157e+308*. Im folgenden Beispiel wird der aufgeführte Wert dynamisch in das Dokument geschrieben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
document.writeln("MAX_VALUE: "+Number.MAX_VALUE);
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 12.27: Ausgabe der größten in JavaScript zulässigen Zahl

Ausgelöst wird der Script-Bereich direkt mit dem Einlesen der Datei. Über **Number.MAX_VALUE** wird der größtmögliche Wert ermittelt. Dessen Ausgabe wird über die **writeln()**-Methode realisiert.

12.3 Die kleinste erlaubte Zahl

Die **MIN_VALUE**-Eigenschaft speichert die kleinste Zahl, die von JavaScript verarbeitet werden kann. Es handelt sich hierbei um den Wert `5e-324`. Anhand der nachstehenden Syntax wird dieser Zahlenwert in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
 var kleinsteZahl = Number.MIN_VALUE;
 alert(kleinsteZahl);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 12.28: Ausgabe der kleinsten in JavaScript zulässigen Zahl

Ausgelöst wird der Script-Bereich direkt mit dem Einlesen der Datei. Hierin wird der Variablen `kleinsteZahl` die Anweisung **Number.MIN_VALUE** zugewiesen. Die Ausgabe von `kleinsteZahl` erfolgt mittels der **alert()**-Methode.

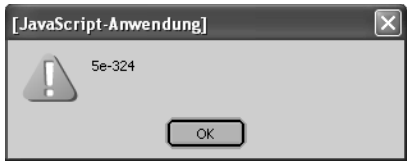


Abbildung 12.7: Die kleinste erlaubte Zahl

12.4 Keine gültige Zahl

NaN = *Not a Number*

Bei der **NaN**-Eigenschaft handelt es sich um ein spezielles Symbol für ungültige Ergebnisse einer Rechnung. **NaN** (Not a Number) eignet sich jedoch nicht zur Überprüfung der Gültigkeit von Zahlen. Vielmehr kann hierdurch einer Zahl die Eigenschaft „nicht gültig" zugewiesen werden. Anhand der nachstehenden Syntax wird der Zeichenfolge `30k` die Eigenschaft „nicht gültig" zugewiesen und dieser Wert wird in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Wert = "30k";
if(Wert != 3) Wert = Number.NaN;
alert(Wert);
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 12.29: Der Wert `30k` ist ungleich 3

Der Variablen `Wert` wird der Wert `30k` zugewiesen. Anschließend wird in Form einer **`if`**-Abfrage überprüft, ob dieser Wert ungleich `3` ist. Ist dies der Fall, und das ist er hier, wird der Variablen `Wert` die Eigenschaft **NaN** zugewiesen. Bei der Ausgabe, die hier mittels der **`alert()`**-Methode realisiert wird, wird nun nicht der Wert `30k`, sondern **NaN** ausgegeben.

12.4.1 Expotentialschreibweise einer Zahl erzwingen

Durch die **`toExpotential()`**-Methode wird die Exponentialschreibweise eines numerischen Wertes erzwungen. So wandelt diese Methode die Zahl 329 in den Wert 3.29e+2 um. Dieses Verhalten wird sich die nachfolgende Syntax zu Nutze machen. Hier wird ein vorgegebener Zahlenwert in Exponentialschreibweise in einem Meldungsfenster angezeigt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Wert = 1034532;
alert(Wert.toExponential());
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 12.30: Die Zahl in Exponentialschreibweise

Ausgelöst wird der Script-Bereich mit dem Einlesen der Datei. Der Variablen `Wert` wird der Wert **`1034532`** zugewiesen. Hieraus ermittelt die **`toExpotential()`**-Methode den Wert *`1.034532e+6`*. Dieser wird mittels der **`alert()`**-Methode ausgegeben.

12.4.2 Bestimmte Anzahl an Nachkommastellen erzwingen

Durch die **`toFixed()`**-Methode kann eine Zahl mit einer exakt definierten Anzahl an Nachkommastellen dargestellt werden. Beachten Sie, dass diese Methode nur dann sinnvoll ist, wenn die entsprechende Zahl auch tatsächlich Nachkommastellen besitzt. Als Parameter wird ein numerischer Wert erwartet, der angibt, wie viele Nachkommastellen die Zahl besitzen soll. Die nachstehende Syntax liest zunächst den Wert `3000139. 992992` ein. Die Darstellung der Nachkommastellen soll auf vier begrenzt werden. Das Ergebnis von `3000139.9930` wird in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Ziffer = 3000139.992992;
alert(Ziffer.toFixed(4));
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 12.31: Es werden nur noch vier Nachkommastellen angezeigt.

Der Script-Bereich wird mit dem Einlesen der Datei ausgelöst. Hierin wird zunächst die Variable `Ziffer` deklariert. Als Wert wird dieser die Zahl `3000139.992992` zugewiesen. Der **`toFixed()`**-Methode wird der Parameter `4` übergeben. Das bedeutet, dass nach der Anwendung der Methode nur noch vier Nachkommastellen angezeigt werden. Der ermittelte Wert wird über die **`alert()`**-Methode ausgegeben.

12.4.3 Genauigkeit einer Zahl bestimmen

Maximalwert ist **21**

Die **`toPrecision()`**-Methode bestimmt, mit welcher Genauigkeit ein Zahlenwert gespeichert werden soll. Als Parameter wird ein numerischer Wert erwartet. Hierbei ist ein Maximalwert von **`21`** möglich. Wird der Wert `232.78` übergeben, liefert die **`toPrecision()`**-Methode bei der definierten Genauigkeit von `3` den Wert `233`. Im folgenden Beispiel werden für einen übergebenen Zahlenwert unterschiedliche Genauigkeiten definiert und die ermittelten Werte werden nacheinander in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
<!--
Zahl = 19933.388883;
alert("Genauigkeit 1 = " + Zahl.toPrecision(1));
alert("Genauigkeit 2 = " + Zahl.toPrecision(2));
alert("Genauigkeit 3 = " + Zahl.toPrecision(3));
alert("Genauigkeit 14 = " + Zahl.toPrecision(14));
alert("Genauigkeit 21 = " + Zahl.toPrecision(21));
//-->
</SCRIPT>
</BODY>
</HTML>
```

Listing 12.32: Die Zahl wird in verschiedenen Genauigkeiten ausgegeben.

Der Variablen `Zahl` wird der Wert `19933.388883` zugewiesen. Dieser Wert wird nun mittels der **`toPrecision()`**-Methode und jeweils unterschiedliche Parameter auf verschiedene Genauigkeiten definiert. Die Rückgabewerte werden mittels der **`alert()`**-Methode ausgegeben.

12.4.4 Eine Zahl in eine Zeichenkette umwandeln

Zahlensystem bekannt geben

Die **toString()**-Methode wandelt eine übergebene Zahl in eine Zeichenkette um. Optional kann ein Parameter angewandt werden. Hierbei handelt es sich um einen numerischen Wert zwischen **2** und **36**, welcher angibt, welches Zahlensystem angewandt werden soll. Im folgenden Beispiel wird die Zahl `255` zweimal ausgegeben. Variante eins verwendet keinen, Variante zwei verwendet einen Parameter. Die ermittelten Werte werden in einem Meldungsfenster ausgegeben.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Wert = new Number(255);
alert(Wert.toString());
alert(Wert.toString(16));
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

Listing 12.33: Der Wert `16` wandelt die Zahl `255` in `ff` um.

Ausgelöst wird der Script-Bereich mit dem Einlesen der Datei. Der Variablen `Wert` wird der Wert `255` zugewiesen. Anschließend wird auf diese Variable zweimal die **toString()**-Methode angewandt. Der ersten **toString()**-Methode wird kein Parameter übergeben. Der ausgegebene Wert ist hier `255`. Bei der zweiten Anwendung wird der Parameter `16` übergeben. Dieser legt fest, dass die Ausgabe anhand des Hexadezimalsystems erfolgt. Die Ausgabe ist hier *ff*.

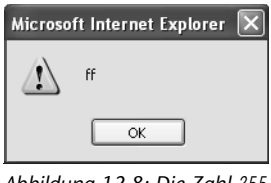


Abbildung 12.8: Die Zahl `255` wurde in die Zeichenkette `ff` umgewandelt.

12.5 Euro-Rechner

auch für andere Währungen geeignet

Nun haben wir den Euro zwar bereits über ein Jahr, dennoch rechnen viele Menschen die Preise noch in DM um. Um all diesen Leuten Rechnung zu tragen, zeigt die hier vorgestellte Syntax, wie eine solche Umrechnung mittels JavaScript realisiert werden kann.

Abbildung 12.9: Der Euro-Rechner in Aktion

Dem Anwender werden beim Laden der Seite zwei Eingabefelder angezeigt. Der im linken Feld eingetragene Wert wird als DM-Angabe gewertet. Im rechten Feld wird der entsprechende Euro-Betrag angezeigt. Als Wechselkurs wird hier `1.95583` angenommen. Das Beispiel ist so konzipiert, dass es auch leicht für andere Währungen genutzt werden kann. Hierzu müssen lediglich der Wechselkurs sowie die Feldbezeichner modifiziert werden.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Kurs = 1.95583;
function Euro()
{
var DMGeld = document.Formular.DM.value;
var EuroGeld = Math.round(100*DMGeld/Kurs)/100;
document.Formular.EURO.value = EuroGeld;
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT name="DM" size="10" onchange="Euro()">DM =
<INPUT name="EURO" size="10">Euro
</FORM>
</BODY>
</HTML>
```

Listing 12.34: Diese Syntax lässt sich auch leicht für andere Währungen modifizieren.

Durch die Variable `Kurs` wird der Wechselkurs angegeben. Die Funktion `Euro()` wird durch den Event-Handler **`onchange`** ausgelöst. Die Variable `DMGeld` speichert den Wert des Eingabefeldes `DM`. Die eigentliche Umrechnung von DM in Euro wird durch die Variable `EuroGeld` durchgeführt. Um einen verständlichen Wert zu erhalten, wird die **`round()`**-Methode des **`Math`**-Objekts verwendet. Diese rundet den erhaltenen Wert kaufmännisch auf oder ab. Als Parameter wird dieser Methode der Ausdruck `100*DMGeld/Kurs` zugewiesen. Hierdurch wird der eingegebene DM-Betrag mit `100` multipliziert und dieser Wert durch den Wert `1.95583` dividiert. Der hier ermittelte kaufmännisch gerundete Wert wird durch `100` dividiert. Im letzten Schritt wird dem Eingabefeld Euro der ermittelte Wert der Variablen `EuroGeld` zugewiesen.

12.6 Gewichtsprüfer

Haben Sie Idealgewicht?

Werfen wir einen Blick auf unser Wohlbefinden und unser Gewicht. Wie lässt sich unser Idealgewicht ermitteln? Ein Arzt kann hier zwar detailliertere Informationen liefern, einen groben Überblick können wir uns aber auch per JavaScript beschaffen. In diesem Abschnitt lernen Sie, wie sich einfache mathematische Berechnungen gezielt nutzen lassen. Mit dem Laden der Seite wird folgender Anblick geboten:

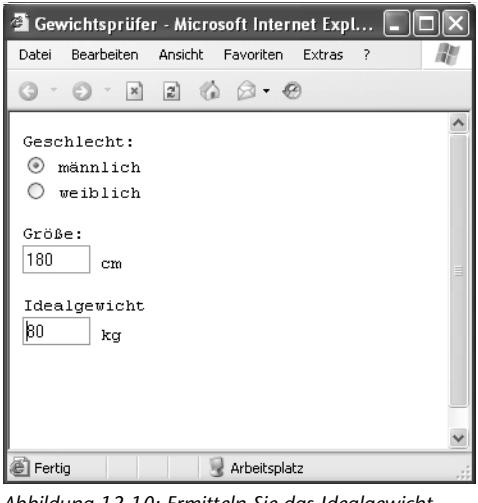


Abbildung 12.10: Ermitteln Sie das Idealgewicht

Der Anwender muss zunächst sein Geschlecht auswählen. Dies ist notwendig, da hiervon die Berechnung des Idealgewichts abhängt. Im nächsten Schritt muss die Größe in cm angegeben werden. Zur Berechnung legen wir zwei einfache Logarithmen zu Grunde. Das männliche Idealgewicht wird durch Größe -100 ermittelt. Für Frauen wird das Idealgewicht

über Gewicht -100*0.9 berechnet. Diese beiden Angaben genügen, um einen kleinen Rechner zur Gewichtsüberprüfung zu programmieren.

```
<HTML>
<HEAD>
  <SCRIPT type="text/JavaScript">
   <!--
    var Geschlecht = "weiblich";
    function Idealgewicht()
    {
     var Groesse = document.Formular.Groesse.value;
     var Gewicht = Groesse - 100;
     if (Geschlecht=="weiblich")
     {
      Gewicht = 0.9 * Gewicht;
     }
     document.Formular.Gewicht.value = Gewicht;
    }
   //-->
  </SCRIPT>
 </HEAD>
 <BODY>
<FORM name="Formular">
<P ALIGN="CENTER">
Geschlecht:
<INPUT type="radio" name="gesch" onclick="Geschlecht='maennlich'"> männlich
<INPUT type="radio" name="gesch" onclick="Geschlecht='weiblich'" checked>
weiblich <BR>
Größe:<INPUT type="text" name="Groesse" size="5" onchange="Idealgewicht()">
cm <BR><BR>
Idealgewicht
<INPUT type="text" name="Gewicht" size="5" onclick="Idealgewicht()"> kg
</p>
</FORM>
</BODY>
</HTML>
```

Listing 12.35: Überprüfen Sie doch einmal Ihr Idealgewicht.

Ausgelöst wird die Funktion entweder durch das Verlassen des Eingabefeldes oder das Anklicken des Formular-Buttons. Die Variable `Geschlecht` dient dazu herauszufinden, ob der Anwender weiblich oder männlich ist. Ist der Wert männlich, wird die Variable `Gewicht` mit dem aus `Groesse - 100` berechneten Wert belegt und als Wert des Eingabefeldes `Gewicht` ausgegeben. Ist der Anwender weiblich, was durch die **`if`**-Abfrage überprüft wird, wird das aus `Groesse - 100` berechnete Gewicht mit dem Wert `0.9` multipliziert. Auch dieser Wert wird in dem einzeiligen Eingabefeld `Gewicht` angezeigt.

12.7 Gerade oder ungerade Zahl

Was für den Menschen eine einfache Aufgabe ist, kann in der Computerwelt ein Problem darstellen. Einen solchen Fall beschreibt das folgende Beispiel. Während es für uns keine Mühe bereitet herauszufinden ob eine Zahl gerade oder ungerade ist, kann sich dies mittels einer Programmiersprache schnell zu einer komplexen Anwendung entwickeln. Betrachten wir zunächst das Ziel unserer Anwendung anhand der Abbildung 12.11. Wie Sie sehen, steht ein einzeiliges Eingabefeld bereit. Hierin wird ein numerischer Wert notiert. In einem Meldungsfenster wird ausgegeben, ob die Zahl gerade oder ungerade ist.

Abbildung 12.11: Die Zahl 5 ist ungerade.

Überlegen wir uns zunächst einen Logarithmus zur Lösung unseres Problems. Jede gerade Zahl muss aus der Division mit 2 eine Ganzzahl zurückliefern. So ergibt dies für die gerade Zahl 6 den Wert 3, da 6 : 2 = 3 ergibt. Somit ist 6 eine gerade Zahl. Was passiert, wenn eine ungerade Zahl durch 2 dividiert wird? Schauen wir uns auch hierzu ein Beispiel an. Teilen wird die Zahl 9 durch 2, ergibt dies den Wert 4.5. Da wir als Ergebnis keine Ganzzahl erhalten, steht fest, dass die Zahl 9 ungerade ist. Die Tatsache, dass die Division einer ungeraden Ganzzahl mit 2 immer einen Wert mit einem Nachkommaanteil von 5 liefert, können wir uns zu Nutze machen. Ein weiteres Problem stellen vom Anwender eingegebene Zahlen mit einem Nachkommaanteil dar. In unserem Beispiel werden solche Zahlen jeweils in eine Ganzzahl umgewandelt. So wird beispielsweise die Zahl 8.9 zunächst in die Ganzzahl 8 umgewandelt. Anschließend wird diese Zahl daraufhin überprüft, ob sie gerade oder ungerade ist.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function GeradeUngerade()
{
var Inhalt = parseInt(document.Formular.Eingabe.value);
var Feld = document.Formular.Eingabe.value;
var Wert = Inhalt / 2;
var Berechnung = Wert.toString();
if(Berechnung.charAt(Berechnung.indexOf('.')+1) == 5)
{
alert('Die Zahl (' + Feld + ') ist eine ungerade Zahl');
}
else
{
alert('Die Zahl (' + Feld + ') ist eine gerade Zahl');
}
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type="button" value="teste Zahl" onclick="GeradeUngerade()">
</FORM>
</BODY>
</HTML>
```

Listing 12.36: Ist die eingegebene Zahl gerade oder ungerade?

Zunächst wird der Wert des Eingabefeldes `Eingabe` durch die Verwendung der **`parseInt()`**-Methode in eine Ganzzahl umgewandelt. Die `Feld`-Variable greift auf den Wert des Eingabefeldes zu. Diesen Wert benötigen wir für die Ausgabe der Zahl. Die in der Variablen `Inhalt` gespeicherte Ganzzahl wird durch `2` geteilt und der Variablen `Wert` zugewiesen. Da wir im Laufe dieses Programms noch Methoden des **`String`**-Objekts verwenden wollen, muss der ermittelte Wert in eine Zeichenkette umgewandelt werden. Dies wird durch die **`toString()`**-Methode realisiert. Die Zeichenkette wird in der Variablen `Berechnung` gespeichert. Wenden wir uns nun der **`if`**-Abfrage zu. Diese fragt ab, ob innerhalb der Zahl ein Punkt vorkommt. Ferner wird überprüft, ob das erste Zeichen nach dem Punkt eine `5` ist. Für den Fall, dass diese Bedingung erfüllt ist, ist die Zahl ungerade. Wird die Bedingung nicht erfüllt, ist sie gerade.

12.8 Fragen und Übungen

1. In ein einzeiliges Eingabefeld soll eine Zahl eingegeben werden können. Hiervon soll die Quadratwurzel ermittelt und in einem Meldungsfenster ausgegeben werden. Die Anzahl der Nachkommastellen ist auf 1 zu begrenzen.

2. Entwickeln Sie ein Programm, das den numerischen Wert eines einzeiligen Eingabefeldes speichert und dessen Potenz in einem Meldungsfenster ausgibt.

3. Schreiben Sie eine Funktion, mit der die Fakultät einer Zahl berechnet werden kann. Wie Sie die Fakultät berechnen können, soll anhand der Zahl 7 gezeigt werden.

4. Beachten Sie, dass die einzugebene Zahl nicht höher als 170 sein darf, da ansonsten der JavaScript-Gültigkeitsbereich verlassen wird. Tipp: Überlegen Sie, welche Schleifenart für diese Anwendung geeignet ist. Die Eingabe der Zahl soll in einem über **`prompt()`** erzeugten Eingabefenster realisiert werden. Die Ausgabe der Zahl soll mittels der **`write()`**-Methode umgesetzt werden.

5. Entwickeln Sie ein Programm, mit dessen Hilfe eine Zufallsgrafik als Hintergrundbild einer Seite angezeigt wird. Die angezeigte Grafik soll beim Laden der Seite gegen eine andere ausgetauscht werden. Begrenzen Sie die Anzahl der möglichen Grafiken auf drei. Tipp: Verwenden Sie die **`random()`**-Methode.

Lösungen

A.1 Antworten zu Kapitel 2

1. Mit welchem Attribut sollten JavaScript-Programme im einleitenden **<SCRIPT>**-Tag gekennzeichnet werden?

   type

2. Wie lautet der MIME-Typ von JavaScript?

   text/javascript

3. Nennen Sie die Aufgabe des **charset**-Attributs.

   Durch den Einsatz des **charset**-Attributs wird der innerhalb einer externen Datenquelle verwendete Zeichensatz spezifiziert.

4. Welches der folgenden ISO-Zeichensätze ist der für den westeuropäischen Sprachraum geeignetste?

   ISO-8859-1

5. Warum führt die folgende Syntax zu keiner Anzeige im Browser, aber auch zu keiner Fehlermeldung?

```
<SCRIPT type="JavaScript">
<!--
document.write("Hallo Welt")>
//-->
</SCRIPT>
```

   Dem **type**-Attribut wurde nicht der korrekte MIME-Typ **text/JavaScript**, sondern lediglich **JavaScript** zugewiesen. Ein WWW-Browser kann diesen MIME-Typ nicht kennen und ignoriert somit den Script-Bereich.

A.2 Antworten zu Kapitel 3

1. Welche der folgenden selbst vergebenen Namen sind korrekt?

   Fotos2

2. Woran erkennen Sie einen Anweisungsblock?

   Anweisungsblöcke bestehen mindestens aus zwei Anweisungen. Mehrere Anweisungen werden durch geschweifte Klammern umschlossen.

A.3 Antworten zu Kapitel 4

1. Ein Programm soll den Text `Hallo Welt` in einem Meldungsfenster ausgeben. Welche(s) Code-Fragment(e) ist (sind) korrekt?

```
<SCRIPT type="text/JavaScript">
<!--
alert("Hallo Welt">
//-->
</SCRIPT>

<SCRIPT type="text/JavaScript">
<!--
document.write("Hallo Welt">
//-->
</SCRIPT>

<SCRIPT type="text/JavaScript">
<!--
var Text = "Hallo Welt";
alert("Text">
//-->
</SCRIPT>
```

   Lediglich das erste Script erfüllt die Aufgabenstellung. Das zweite Script gibt den Text nicht in einem Meldungsfenster aus, sondern schreibt ihn dynamisch in das Dokument. Beim letzten Script wird zwar eine Variable deklariert, da diese jedoch innerhalb der **`alert()`**-Methode in Anführungszeichen gesetzt wird, gibt der Browser nicht den Text `Hallo Welt`, sondern `Text` aus.

2. Welcher der folgenden Event-Handler ist in JavaScript nicht bekannt?

   `onTime`

A.4 Antworten zu Kapitel 5

1. Welches sind gültige Eigenschaften des **window**-Objekts?

   closed
   statusbar

2. Schreiben Sie eine Funktion, durch die beim Einlesen einer Seite ein zweites Fenster geöffnet wird. Im Hauptfenster soll sich ein Verweis befinden. Durch dessen Anklicken soll in einem Meldungsfenster ausgegeben werden, ob das neue Fenster noch offen ist oder nicht.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Zweitfenster = window.open("neu.htm");
function pruefeFenster()
{
 if(Zweitfenster.closed == true) alert("nein");
 else alert("ja");
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:pruefeFenster()">Fenster auf?</A>
</BODY>
```

A.5 Antworten zu Kapitel 6

1. Worin besteht u.a. der Unterschied zwischen dem **document**- und dem **window**-Objekt?

   Das **document**-Objekt bezieht sich auf den Inhalt, der in einem Browserfenster angezeigt wird. Über das **window**-Objekt kann das Browserfenster kontrolliert, neu erzeugt und abgefragt werden. In der Objekthierarchie ist **window** das oberste Objekt.

2. Ändern Sie die Syntax aus Abschnitt 6.5 so um, dass nicht die Farbe der aktiven Hyperlinks, sondern der Hintergrund des Dokuments geändert wird.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
function wechselFarbe()
{
```

```
 if(document.Formular.test[0].checked == true)
 document.bgColor = "red";
 else if(document.Formular.test[1].checked == true)
 document.bgColor = "blue";
 else document.bgColor = "000000";
}
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="radio" value="rot" name="test">
<INPUT type="radio" value="blau" name="test">
</FORM>
<A href="javascript:wechselFarbe()">Hintergrund &auml;ndern</A>
</BODY>
</HTML>
```

A.6 Antworten zu Kapitel 7

1. Fügen Sie in das Beispiel auf Seite 249 eine neue Funktion ein, mit deren Hilfe nach vorne geblättert werden kann.

```
function zurueck(Frame1,Frame2)
{
  rechtsoben=eval("parent."+Frame1);
  rechtsunten=eval("parent."+Frame2);
  rechtsoben.history.forward();
  rechtsunten.history.forward();
}
```

2. Welche Eigenschaft, die im **frames**-Objekt einsetzbar ist, kann im **window**-Objekt nicht eingesetzt werden?

   length

A.7 Antworten zu Kapitel 8

1. Schreiben Sie eine Funktion, mit der Sie einen Browser der 4. und der 5. Generation erkennen können. Die Browsergeneration soll in einem Meldungsfenster ausgegeben werden. Tipp: Um die Browsergeneration zu ermitteln, müssen Sie mittels der **substring()**-Methode eine Teilzeichenkette aus dem zurückgelieferten Wert extrahieren. Die **substring()**-Methode erwartet zwei Parameter. Der erste gibt die Position des ersten zu extrahierenden Zeichens und der zweite die Position des ersten nicht mehr zu extrahierenden Zeichens an. Finden Sie eine geeignete

Methode, um die Produktversion des Browsers zu erfahren. Erst anschließend sollten Sie mit der Extrahierung beginnen.

```
<HTML>
<HEAD>
<SCRIPT type="text/javascript">
<!--
if(navigator.appVersion.substring(0,1) == "4")
 alert("Browsergeneration 4");
else if (navigator.appVersion.substring(0,1) == "5")
alert("Browsergeneration 5");
else ("Schade");
//-->
</SCRIPT>
</HEAD>
<HEAD>
</BODY>
</HTML>
```

2. Warum ist die **appCodeName**-Eigenschaft des **navigator**-Objekts kein geeignetes Mittel, um Browserweichen zu erstellen?

   Sowohl der Netscape Navigator, der Internet Explorer als auch Opera speichern in dieser Eigenschaft den Wert „Mozilla“. Eine Unterscheidung der jeweiligen Browser ist somit nicht gewährleistet.

3. Es ist ein Programm zu entwickeln, durch welches alle verfügbaren Dateiendungen, die vom Netscape Navigator im Zusammenhang mit MIME-Typen interpretiert werden können, interpretiert werden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<SCRIPT type="text/JavaScript">
for(i=0; i<navigator.mimeTypes.length; ++i)
document.write("<BR><i>" + navigator.mimeTypes[i].suffixes);
</SCRIPT>
</BODY>
</HTML>
```

A.8 Antworten zu Kapitel 9

1. Schreiben Sie ein JavaScript-Programm, welches ein Formular nach 30 Sekunden automatisch abschickt. Tipp: Verwenden Sie die **setTimeout()**-Methode des **window**-Objekts und setzen Sie hier als zweiten Parameter den Wert `30000` für 30 Sekunden.

```
<HTML>
<HEAD>
</HEAD>
<BODY>
<FORM name="Formular" action="../cgi-bin/pruefe.pl" method="get">
<INPUT type="text" name="Name">
</FORM>
<SCRIPT type="text/javascript">
<!--
function automatischesFormular()
 {
  document.Formular.submit();
 }
 window.setTimeout("automatischesFormular()",30000);
//-->
</SCRIPT>
</BODY>
</HTML>
```

A.9 Antworten zu Kapitel 10

1. Schreiben Sie ein Programm, durch das ein Grafikwechsel realisiert werden kann. Die Auswahl der Grafiken soll aus einer Auswahlliste möglich sein. Konzipieren Sie das Programm so, dass die Grafiken unterhalb der Auswahlliste angezeigt werden. Ausgelöst wird der Bilderwechsel durch den Event-Handler **onchange**.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function BilderDropDown()
{
var Eintrag = document.Formular.Grafiken.selectedIndex;
document.images.Bild.src=
document.Formular.Grafiken.options[Eintrag].value
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<SELECT name="Grafiken" onchange=BilderDropDown()>
<OPTION value="linseneintopf.jpg">Grafik 1</OPTION>
<OPTION value="lauchcremesuppe.jpg">Grafik 2</OPTION>
<OPTION value="beutelsuppe.jpg">Grafik 3</OPTION>
</SELECT>
```

```
</FORM>
<IMG src="linseneintopf.jpg" name="Bild">
</BODY>
</HTML>
```

2. SlideShows ermöglichen das wechselseitige Anzeigen unterschiedlicher Grafiken nach einem festgelegten zeitlichen Zyklus. Eingesetzt wird dieses Stilmittel beispielsweise bei Präsentationen oder als ein netter Effekt auf Internetseiten. Um eine SlideShow zu realisieren, sollten alle einzubindenden Grafiken in der gleichen Größe vorliegen. Hierdurch lassen sich stilistisch saubere Präsentationen erstellen. In unserem Beispiel sollen vier Grafiken in einer Geschwindigkeit von 2.000 Millisekunden wechselseitig angezeigt werden. Bereits beim Laden der Seite soll eine der vier Grafiken präsentiert werden. Tipp: Legen Sie alle Grafiken in einem Array ab. Verwenden Sie die **`setTimeout()`**-Methode, um die entsprechende JavaScript-Funktion alle 2.000 Millisekunden neu aufzurufen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Geschwindigkeit = 2000
var Grafik = new Array()
Grafik[0] = 'grafik1.gif'
Grafik[1] = 'grafik2.gif'
Grafik[2] = 'grafik3.gif'
Grafik[3] = 'grafik4.gif'
var j = 0
var Anzahl = Grafik.length
var ladeGrafiken = new Array()
for (i = 0; i < Anzahl; i++){
   ladeGrafiken[i] = new Image()
   ladeGrafiken[i].src = Grafik[i]
}
function SlideShow(){
   document.images.slider.src = ladeGrafiken[j].src
   j = j + 1
   if (j > (Anzahl-1)) j=0
   t = setTimeout('SlideShow()',Geschwindigkeit)
}
//-->
</SCRIPT>
<BODY onload="SlideShow()">
<IMG src="grafik1.gif" name="slider" width="300" height="200">
</BODY>
</HTML>
```

A.10 Antworten zu Kapitel 11

1. Schreiben Sie ein JavaScript-Programm, welches die noch verbleibenden Sekunden bis zum Jahr 2005 in einem Meldungsfenster ausgibt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function verbleibendeSekunden() {
 var Datum = new Date();
 var berechneZeit = Datum.getTime() / 1000;
 var Zieljahr = new Date(2020,0,1,0,0,0);
 var Endpunkt = Zieljahr.getTime() / 1000;
 var Sekunden = Math.floor(Endpunkt - berechneZeit);
 alert(Sekunden);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<A href="javascript:verbleibendeSekunden()">Sekunden</A>
</BODY>
</HTML>
```

2. Schreiben Sie ein Programm, welches den Anwender tageszeitabhängig begrüßt. Der Begrüßungstext soll in das Dokument geschrieben werden. Die Einteilung der Begrüßungen soll in vier Intervallen erfolgen. Überlegen Sie sich vor der Realisierung des Programms günstige Zeitintervalle.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
var Datum = new Date();
var Stunden = Datum.getHours();
if(Stunden >= 6 && Stunden < 12) document.write("Guten Morgen!");
else if(Stunden >= 12 && Stunden < 18) document.write("Guten Tag!");
else if(Stunden >= 18 && Stunden <= 23) document.write("Guten Abend!");
else if(Stunden >= 0 && Stunden < 6) document.write("Gute Nacht!");
//-->
</SCRIPT>
</HEAD>
<BODY>
</BODY>
</HTML>
```

A.11 Antworten zu Kapitel 12

1. In ein einzeiliges Eingabefeld soll eine Zahl eingegeben werden können. Hiervon soll die Quadratwurzel ermittelt und in einem Meldungsfenster ausgegeben werden. Die Anzahl der Nachkommastellen ist auf 1 zu begrenzen.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Quadratwurzel()
{
var Feld = document.Formular.Eingabe.value;
var Ausgabe = Math.sqrt(Feld);
var Genauigkeit = Ausgabe.toPrecision(2);
alert(Genauigkeit);
}
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
<INPUT type="text" name="Eingabe">
<INPUT type=button value="Quadratwurzel" onclick="Quadratwurzel()">
</FORM>
</BODY>
</HTML>
```

2. Entwickeln Sie ein Programm, das den numerischen Wert eines einzeiligen Eingabefeldes speichert und dessen Potenz in einem Meldungsfenster ausgibt.

```
<HTML>
<HEAD>
<SCRIPT type="text/JavaScript">
<!--
function Potenz()
   {
   var Eingabe = document.Formular.Eingabe.value;
   var Ergebnis = Eingabe * Eingabe;
   alert(Ergebnis);
   }
//-->
</SCRIPT>
</HEAD>
<BODY>
<FORM name="Formular">
```

```
<INPUT type="text" name="Eingabe">
<INPUT type=button value="Potenz" onclick="Potenz()">
</FORM>
</BODY>
</HTML>
```

3. Schreiben Sie eine Funktion, mit der die Fakultät einer Zahl berechnet werden kann. Wie Sie die Fakultät berechnen können, soll anhand der Zahl 7 gezeigt werden.

```
f(7) = 1*2*3*4*5*6*7 = 5040
```

4. Beachten Sie, dass die einzugebene Zahl nicht höher als 170 sein darf, da ansonsten der JavaScript-Gültigkeitsbereich verlassen wird. Tipp: Überlegen Sie, welche Schleifenart für diese Anwendung geeignet ist. Die Eingabe der Zahl soll in einem über **`prompt()`** erzeugten Eingabefenster realisiert werden. Die Ausgabe der Zahl soll mittels der **`write()`**-Methode umgesetzt werden.

```
<HTML>
<HEAD>
    <SCRIPT type="text/JavaScript">
     function Fakultaet(x)
     {
       var y = 1;
       var z = x;
       x--;
       if (x != 0)
       {
          y = Fakultaet(x);
       };
       return(y*z);
     }
     var a = 0;
     while(a<1)
     {
       a = prompt("Geben Sie eine Zahl zwischen 1 und 170 ein:","Zahl");
       a = parseInt(a)
     };
     document.write("Die Fakultät von " + a + " = " + Fakultaet(a));
    </SCRIPT>
  </BODY>
</HTML>
```

5. Entwickeln Sie ein Programm, mit dessen Hilfe eine Zufallsgrafik als Hintergrundbild einer Seite angezeigt wird. Die angezeigte Grafik soll beim Laden der Seite gegen eine andere ausgetauscht werden. Begren-

zen Sie die Anzahl der möglichen Grafiken auf drei. Tipp: Verwenden Sie die **random()**-Methode.

```
<HTML>
<HEAD>
</HEAD>
<SCRIPT type=text/javascript>
<!--
var Grafik1="lauchcremesuppe.jpg"
var Grafik2="linseneintopf.jpg"
var Grafik3="suppe.jpg"
var Wechsel=Math.round(6*Math.random())
if (Wechsel<=1)
Grafik=Grafik1
else if (Wechsel<=4)
Grafik=Grafik2
else
Grafik=Grafik3
document.write('<BODY background="'+Grafik+'">')
//-->
</SCRIPT>
</BODY>
</HTML>
```

Stichwortverzeichnis

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

Symbole

Numerisch