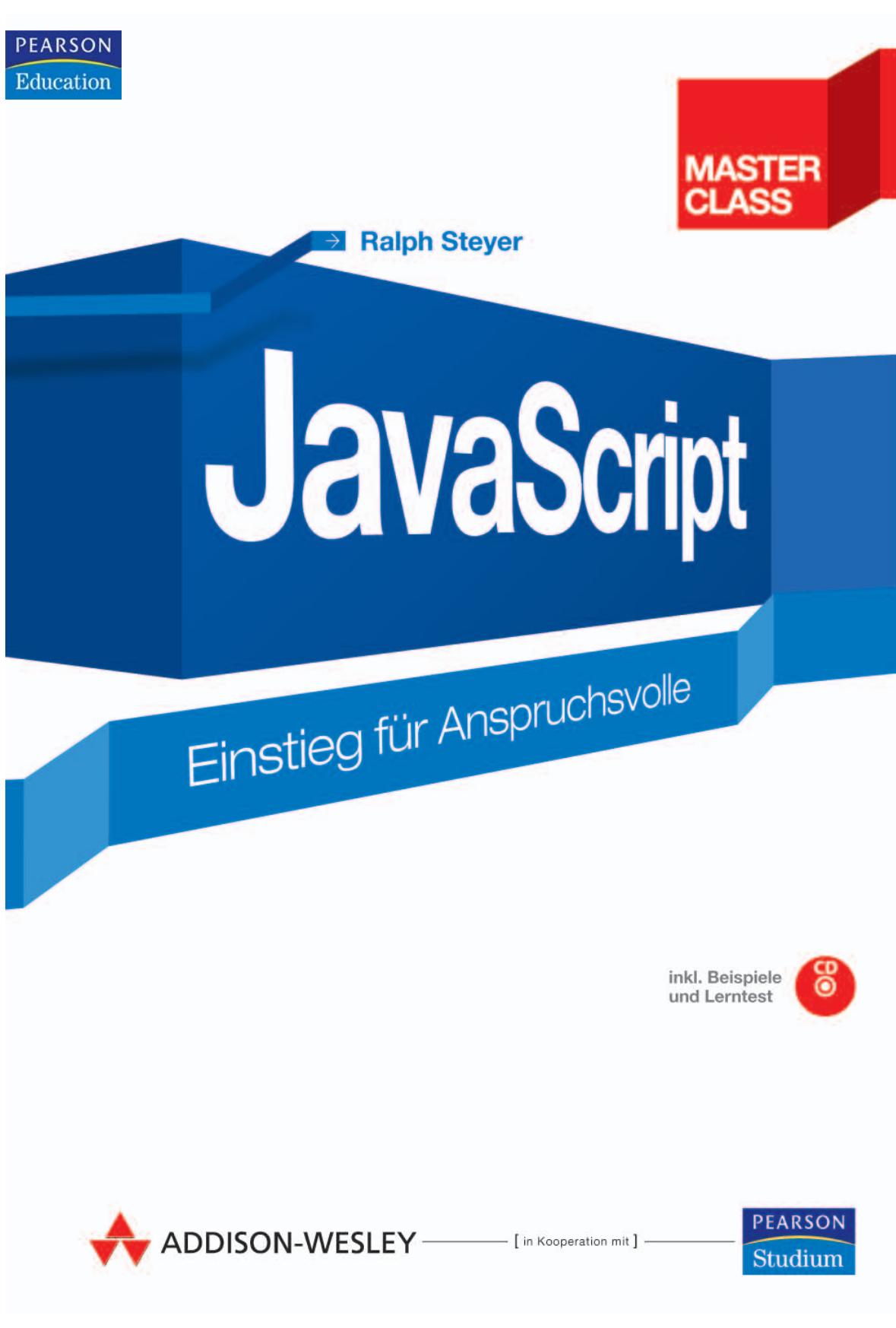

PEARSON
Education
MASTER
CLASS
Ralph Steyer
JavaScript
Einstieg für Anspruchsvolle
inkl. Beispiele
und Lerntest
CD
ADDISON-WESLEY
[in Kooperation mit]
PEARSON
Studium

4

JavaScript in Webseiten einbinden

Bevor Sie mit JavaScript einen Browser beziehungsweise die Webseite programmieren können, müssen Sie eine Verbindung zwischen HTML beziehungsweise XHTML (dem Gerüst der Webseite) und der Skriptfunktionalität schaffen. In diesem Kapitel beschäftigen wir uns mit dieser Verbindung von JavaScript zu einer Webseite. Dies ist ausschließlich Aufgabe von (X)HTML-Befehlen. Obwohl dies vielen Leuten, die sich nur gelegentlich mit JavaScript beschäftigen, als eine Trivialität erscheinen mag, sind hier unzählige Feinheiten zu beachten, die einem JavaScript-Programmierer das Leben schwer machen können. Auf der einen Seite werden diese Feinheiten für 95% aller Fälle wohl keine Rolle spielen, aber es gibt durchaus kritische Situationen, die man nur durch die richtige Einbindung von JavaScripten in die Webseite entschärfen kann.

Sie lernen in diesem Kapitel

- die Inline-Referenz,
- die Referenz auf eine externe JavaScript-Datei,
- das Einbinden eines Skriptcontainers in die Webseite,
- die Festlegung der JavaScript-Version,
- einen Test, ob bei einem Browser JavaScript aktiviert ist,
- einen Test, welche JavaScript-Version von einem Browser unterstützt wird, und
- eine Technik, die sicherstellt, dass ein Browser nur solche JavaScript-Anweisungen ausführt, die er auch versteht,

kennen.

4.1 Einbindung 1 – die Notation eines Skriptcontainers in der Webseite

JavaScript-Anweisungen können in eine Webseite eingebunden werden, indem sie einfach in die entsprechende (X)HTML-Datei als Klartext hineingeschrieben werden. Allerdings muss der Skriptbereich klar von der »normalen« Webseite getrennt werden[1]. Der Beginn eines Skripts wird durch eine (X)HTML-Steueranweisung realisiert, die mit ihrem zugehörigen Abschluss-Tag einen Container für die Skriptanweisungen bildet.

Es handelt sich bei dieser Steueranweisung um den Tag `<script>`. Dieser gehört explizit zu (X)HTML, aber all das, was in dem eingeschlossenen Bereich notiert wird, wird vom Browser als ein Skript interpretiert. Über den optionalen Parameter `language` können Sie angeben, um welche Skriptsprache es sich handelt. Der Token `JavaScript` steht dabei natürlich für die Skriptsprache JavaScript. Aber hier kann man sowohl andere Sprachen angeben als auch noch genauere Angaben machen, auf die wir noch eingehen.

Im gesamten Tag ist im Fall von HTML die Groß- und Kleinschreibung vollkommen **irrelevant**, denn es handelt sich bei allen Angaben innerhalb der spitzen Klammern um reines HTML. Das bedeutet, die Anweisungen `<script language="JavaScript">` oder `<Script Language="javascript">` sind vollkommen äquivalent. Ebenso können Sie bei reinem HTML auf die Hochkommata bei der Wertzuweisung verzichten (`<script language=javascript>`), obwohl die strenge Auslegung der HTML-Syntax diese im Prinzip fordert.

Im Fall von XHTML sind wir bezüglich der Groß- und Kleinschreibung nicht ganz so frei. In diesem Fall müssen der Tag und das Attribut auf jeden Fall kleingeschrieben werden. Ebenso muss der Wert von `language` auf jeden Fall in Hochkommata gesetzt werden. Die Groß- und Kleinschreibung bei dem Token `JavaScript` als zugewiesenen Wert ist jedoch wieder vollkommen irrelevant.

Tipp

Sie sollten sich auf jeden Fall die XHTML-konforme Notation angewöhnen.

4.1.1 Der type-Parameter

In dem `<script>`-Tag gibt es noch weitere optionale Attribute wie `type`. Damit geben Sie den MIME-Typ an (für JavaScript ist das der Wert `"text/javascript"`). Auf diesen Parameter können Sie normalerweise **verzichten** und sollten es in gewissen Konstellationen sogar tun. Es ist nämlich so, dass es ein explizites Problem mit diesem Attribut gibt, auf das in den Ausführungen zur Angabe der JavaScript-Version eingegangen werden soll (*siehe Seite 139*).

1. So wie bei Style Sheets.

Hinweis

Diese Empfehlung ist umso bemerkenswerter, da die `type`-Angabe der offizielle Standard des W3C ist und sie damit explizit den Vorgabe W3C widerspricht. Das Attribut `language` gilt nach offiziellen Vorgaben als deprecated, ist aber in der Praxis das Attribut, das Sie verwenden sollten.

Wenn Sie eine andere Skriptsprache oder -angabe verwenden wollen (etwa VBScript, JScript oder auch der ursprüngliche Name von JavaScript – LiveScript), geben Sie diese vollkommen analog an. Entweder über den MIME-Typ oder eben das `language`-Attribut. Beachten Sie, dass die Angabe von JScript oder LiveScript künstlich Probleme in einigen Browsern erzeugen kann, die mit der Angabe von JavaScript als Sprache zurechtkommen würden. Es gibt wenige Situationen, wo das sinnvoll ist. Hingegen ist die Angabe der JavaScript-Version unter manchen Umständen zweckmäßig (*siehe auf Seite 139*).

Hinweis

Im Rahmen der Beispiele in dem Buch wird weitgehend auf die explizite Auszeichnung der JavaScript-Version verzichtet. Ausnahme sind die hier folgenden Rezepte, die genau dieses Thema behandeln, sowie diejenigen Rezepte, in denen explizit Techniken jenseits von JavaScript 1.3 eingesetzt werden oder wo der besprochene Effekt auf der Angabe der JavaScript-Version beruht.

Sie können (und werden in vielen Fällen) in einer Webseite **mehrere Skriptcontainer** verwenden (sogar mit verschiedenen Skriptsprachen).

Ebenso können Sie auf das Attribut `language` bzw. den MIME-Typ ganz verzichten. In dem Fall wird der Default-Skriptinterpreter des Browsers aufgerufen. Dies ist in allen bekannten Fällen der JavaScript-Interpreter, aber darauf verlassen sollte man sich nicht. Besser geben Sie in der Praxis die Skriptsprache explizit an.

Ein `<script>`-Tag kann an jeder beliebigen Stelle innerhalb einer Webseite platziert werden. Die Skriptanweisungen werden einfach geladen, wenn die Webseite von oben nach unten im Browser abgearbeitet wird. Wenn der Browser eine auszuführende Anweisung lädt, wird sie ausgeführt.

Entsprechend ist klar, dass unten in einer Webseite notierte Skriptanweisungen auch erst dann zur Verfügung stehen, wenn die Seite bis dahin geladen ist. Das ist dann kein Problem, wenn das Skript genau da notiert wird, wo es benötigt wird und es beim Laden automatisch ausgeführt werden soll.

Hinweis

So künstlich oder antik die Situation Ihnen vielleicht erscheinen mag – es gibt auch heute noch einige Anwendungen, bei denen man einen Skriptcontainer weit unten in die Webseite (sogar nach dem Body) notiert. Es wird dann gemacht, wenn man in einem Skript auf Bestandteile der Webseite zugreifen will und gewährleisten muss, dass diese bereits im Rahmen des DOM- Konzeptes als Objekte zur Verfügung stehen.

Zwar gibt es die Möglichkeit, mit Eventhandlern zu arbeiten, die erst nach dem Laden der Webseite ausgeführt werden. Die Erfahrung zeigt jedoch, dass es einige Situationen gibt, in denen dies sehr unzuverlässig funktioniert. Selbst das moderne Atlas-Framework (ein Microsoft-Framework zur Unterstützung von AJAX-Applikationen) benötigt – zumindest in einigen Betaversionen – für einige Anwendungen diesen Trick, damit die Bestandteile einer Webseite als Objekte für den JavaScript-Zugriff zur Verfügung stehen. Das Google Web Toolkit (ein weiteres Framework zur Generierung von AJAX-Applikationen – hier aber aus serverseitigem Java) bindet dagegen ein externes Skript explizit als erste Anweisung innerhalb des Dokumentenköpers ein, da dies nach Aussage von Google für sie extreme Vorteile bei der Performance bringt.

In JavaScript ist es allgemein üblich, mit Funktionen bestimmte Skriptpassagen zusammenzufassen und vor dem automatischen Ausführen beim Laden zu schützen. JavaScript-Funktionen werden erst ausgeführt, wenn sie aufgerufen werden. Das verzögerte Ausführen ist keineswegs ungewöhnlich. Das ist in fast allen Situationen, wo Skripte eingesetzt werden sollen, sinnvoll. Das automatische Ausführen eines Skripts beim Laden der Webseite ist die Ausnahmesituation. Meist ist es so, dass ein Skript beim Klick auf einen Button oder Hyperlink, beim Überstreichen eines Bereichs mit der Maus, beim Verlassen einer Webseite oder einem ähnlichen Ereignis ausgeführt werden soll. Wenn nun der Aufruf eines Skripts erfolgt, bevor das Skript selbst geladen wurde, hat man ein Problem.

Eine übliche Vorgehensweise ist es deshalb, einen `<script>`-Tag entweder sehr weit oben in der Webseite (etwa innerhalb des `<head>`-Abschnitts eines Dokuments, davor oder direkt dahinter) zu platzieren oder bei der automatischen Ausführung an der Stelle, wo er gebraucht wird. Wenn in dem Skriptcontainer Funktionen oder Variablen auftauchen, merkt sich der Browser diese, und sie sind dann für das ganze Dokument verfügbar. Der Code wird so lange verfügbar sein, wie das Dokument im Browserfenster geladen ist.

4.1.2 Der HTML-Kommentar im Skriptbereich

Bei einem Skriptcontainer stand in der Vergangenheit in der Regel direkt nach dem einleitenden `<script>`-Tag ein HTML-Kommentar (`<!--`). Dieser wird unmittelbar vor dem abschließenden Tag `</script>` mit dem entsprechenden HTML-Kommentar-Endezeichen (`-->`) wieder geschlossen. Dadurch steht der gesamte JavaScript-Code innerhalb eines HTML-Kommentars.

Hinweis

Dieses Verfahren zur Absicherung von alten Browsern ist heute aber nicht mehr zwingend. Es ist zwar sicherer für den Fall, dass ein Browser die Seite lädt, der keine Skripte interpretieren kann. Aber diese Browser stammen aus den frühen neunziger Jahren und sollten wirklich ausgestorben sein.

Beachten Sie, dass hier nicht von Browsern die Rede ist, bei denen der Anwender JavaScript deaktiviert hat. Diese verstehen den `<script>`-Tag und führen einfach die darin enthaltenen Anweisungen nicht aus. Die HTML-Kommentare sind ausschließlich für solche Browser gedacht, die keinen `<script>`-Befehl verstehen.

Hinweis

Die Anwendung des HTML-Kommentars im Inneren des `<script>`-Containers beinhaltet einige Fallen. Der Netscape Navigator hatte in einigen älteren Versionen Probleme mit dem Ende-Zeichen des HTML-Kommentars. Er versteht es als JavaScript-Anweisung (`--` ist in JavaScript der Dekrementoperator, und den versucht der Browser auszuführen). Man versteckte daher das Ende des HTML-Kommentars hinter der Zeichenfolge `//`, was einen JavaScript-Kommentar bedeutet.

Aber auch der Explorer hat ein spezifisches Problem, das er mit dem Navigator sowie einigen weiteren Browsern teilt. Wenn die erste JavaScript-Anweisung in der gleichen Zeile wie der HTML-Kommentar-Tag steht, wird diese ignoriert.

Die erste JavaScript-Anweisung darf also nicht (!) in der gleichen Quelltextzeile stehen wie der Beginn des HTML-Kommentars. Der abschließende und hinter einem JavaScript-Kommentar versteckte HTML-Kommentar kann hingegen in der gleichen Quelltextzeile stehen wie die letzte JavaScript-Anweisung. Übersichtlicher ist es jedoch, wenn dieses Ende in einer eigenen Zeile steht.

4.1.3 Ein korrekter Skript-Container

Ein korrekter Skript-Container mit HTML-Kommentaren sieht also vollständig so aus:

```
<script language="JavaScript">
<!--
       [Skript-Anweisungen]
//-->
</script>
```

Listing 4.1
Ein typischer Container zum Einbinden eines JavaScripts (ohne Versionsangabe)

Beispiel (*skriptcontainer.html*):

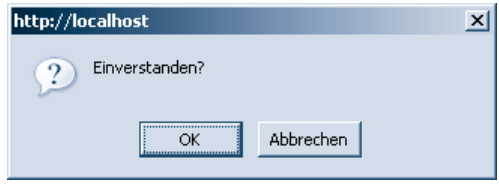
Listing 4.2 Verwendung eines Skriptcontainers

```
01 <html>
02 <body>
03 <script language="JavaScript">
04   if(confirm("Einverstanden?")) document.write("<h1>GUT</h1>");
05   else document.write("<h1>NICHT GUT</h1>");
06 </script>
07 </body>
08 </html>
```

In den Zeilen 3 bis 6 finden Sie einen Skript-Container und die eigentlichen JavaScript-Anweisungen stehen in den Zeilen 4 und 5. Es handelt sich um eine Entscheidungsstruktur, bei der je nach Situation einer von zwei Programmflüssen ausgewählt wird.

Wenn Sie die Datei ausführen, wird ein kleines Dialogfenster angezeigt, in dem der Anwender einen OK- oder einen ABBRECHEN-Button auswählen kann (dies bewirkt die *confirm()*-Anweisung).

Abbildung 4.1 Die Anweisung confirm() zeigt einen Auswahldialog an.



Je nach Auswahl wird einer der beiden angegebenen Texte in die Webseite geschrieben.

4.2 Einbindung 2 – die Verwendung von externen JavaScript-Dateien

In fast jedem Fall macht es sehr viel Sinn, JavaScripts in einer oder mehreren externen Datei(en) auszulagern. Diese Vorgehensweise trägt der Tatsache Rechnung, dass man ab einer gewissen Größe Projekte strukturieren muss. Zudem können gemeinsame Funktionalitäten in einer Datei bereitgestellt werden, und nicht jede Webseite muss deren Implementierung enthalten. Natürlich wird die Wartung und Anpassung erheblich erleichtert, und auch sonst gibt es weitere Vorteile. Die Auslagerung in eine externe Datei funktioniert seit HTML 4 bzw. JavaScript ab der Version 1.1. Es wird bei der Referenz auf externe Skriptdateien wieder der `<script>`-Tag verwendet, der dann aber über das Attribut `src` die externe Datei angibt. Die Syntax ist so:

Listing 4.3 Referenz auf eine externe JavaScript-Datei

```
<script language="JavaScript" src=
"[externe JavaScript-Datei]"></script>
```

In Anführungszeichen wird hinter dem Attribut `src` der Name bzw. vollständige URL der separaten Datei angegeben. Dabei gelten beim Referenzieren von separaten JavaScript-Dateien die üblichen Regeln für URLs. Entweder geben

Sie einen relativen Pfad an oder aber einen vollständigen URL, der mit dem Protokoll (in der Regel `http`) beginnt. Die Datei mit dem Quellcode muss – wie HTML-Dateien – eine reine Klartextdatei sein und ausschließlich JavaScript-Code enthalten. Eine solche JavaScript-Datei enthält explizit kein Grundgerüst. Üblich ist die Dateierweiterung *.js*, aber das ist nicht zwingend. Beispiele:

Listing 4.4
Angabe einer relativ zur Webseite im gleichen Verzeichnis befindlichen JavaScript-Datei

```
<script language="JavaScript" src="meineScripte.js"></script>
```

Listing 4.5
Angabe einer relativ zur Webseite im parallelen Verzeichnis scripte notierten JavaScript-Datei

```
<script language="JavaScript"
  src="../scripte/meineScripte.js"></script>
```

Listing 4.6
Angabe von einem absoluten URL der JavaScript-Datei

```
<script language="JavaScript"
  src="http://rjs.de/scripte/meineScripte.js"></script>
```

Achtung

Bei der Referenz auf externe JavaScript-Dateien sollte der `<script>`-Container auf jeden Fall **leer** bleiben. Die JavaScript-Anweisungen befinden sich alle in der referenzierten externen Datei. Sie können keine JavaScript-Anweisungen in den Container schreiben. Genau genommen muss man diese Aussage aber etwas präzisieren.

Der Skriptcontainer ist im Sinne von XML ein **leeres Element**, wenn man ihn zum Referenzieren einer externen JavaScript-Datei verwendet. Deshalb darf im Inneren streng genommen kein Whitespace-Zeichen (Leerzeichen, Tabulatoren, Zeilenumbrüche etc.) stehen. Durch das Prinzip der Fehlertoleranz verhalten sich allerdings die meisten Browser so, dass sie einen Zeilenumbruch oder ein Leerzeichen im Inneren des Elements nicht als Fehler interpretieren. Vermeiden Sie diese »Verschmutzung« dennoch – es gibt ja keinen Grund dazu.

Aber ein anderes Problem tritt vor allem im Internet Explorer auf. Wenn Sie die vollkommen korrekte XML-Syntax `<script language="JavaScript" src="..." />` zur Angabe eines leeren Elementes verwenden, wird der Internet Explorer einen Fehler melden bzw. die Verwendung der externen JavaScript-Datei verweigern[1]. Sie sollten zur Sicherheit auf diese elegante und wie gesagt vollkommen korrekte Syntax verzichten und stattdessen die oben gezeigte Schreibweise wählen.

Die externe JavaScript-Datei enthält wie ein in einer Webseite notierter `<script>`-Container einfach JavaScript-Code in Form von Klartext. Allerdings wird bei externen JavaScript-Dateien fast immer mit Funktionen gear-

1. Selbst in der neuen Version 7.

beitet, da sonst alle in der externen Datei notierten Anweisungen beim Laden einer Webseite mit einer Referenz auf die externe Datei abgearbeitet werden. Das ist in der Regel nicht gewünscht, denn meist finden sich mehrere Funktionalitäten in einer einzigen Datei, und Funktionen geben die Möglichkeit, gezielt eine der Schrittfolgen aufzurufen.

Auch wenn wir jetzt wieder ein wenig vorgreifen müssen, soll ein praktisches Beispiel nicht fehlen. Hier sehen Sie zuerst die HTML-Datei *externesskript.html* mit der Referenz auf die externe JavaScript-Datei *meineFunktionen.js*, die sich im gleichen Verzeichnis wie die HTML-Datei befinden muss:

Listing 4.7
Die Webseite, in der die externe JavaScript-Datei referenziert wird

```
01 <html>
02 <script language="JavaScript" src="meineFunktionen.js"></script>
03 <body>
04  <h1>Meine Homepage</h1>
05  <script language="JavaScript">
06   willkommen();
07  </script>
08  </body>
09 </html>
```

In der Zeile 2 befindet sich die Referenz auf die externe JavaScript-Datei mit Namen *meineFunktionen.js*. Diese müssen wir natürlich noch erstellen. In Zeile 3 beginnt der normale Body-Bereich der Webseite. In Zeile 4 finden Sie eine gewöhnliche HTML-Überschrift der Ordnung 1. In den Zeilen 5 bis 7 befindet sich ein weiterer Skript-Container, in dem per direkter Einbindung eine JavaScript-Funktion aufgerufen wird (Zeile 6). Diese ist in der externen JavaScript-Datei definiert und die erstellen wir jetzt.

Dies ist das Listing der externen JavaScript-Datei mit Namen *meineFunktionen.js*:

Listing 4.8
Der Inhalt der externen JavaScript-Datei

```
01 function willkommen(){
02   document.write("<h2 align='center'>Willkommen</h2>");
03 }
```

In der externen JavaScript-Datei finden Sie die Deklaration (Definition) einer Funktion mit Namen `willkommen()`. Diese wird in der HTML-Datei aufgerufen. Die Funktion selbst ist sehr einfach. Es wird nur ein einfacher Text samt einigen HTML-Formatierungen (hier eine zentrierte Überschrift der Ordnung 2) mit der JavaScript-Anweisung `document.write()` in die Webseite geschrieben.

Wenn Sie die HTML-Datei in Ihrem Browser aufrufen, wird dieser eine teils fest per HTML, teils dynamisch mit JavaScript geschriebene Webseite anzeigen.

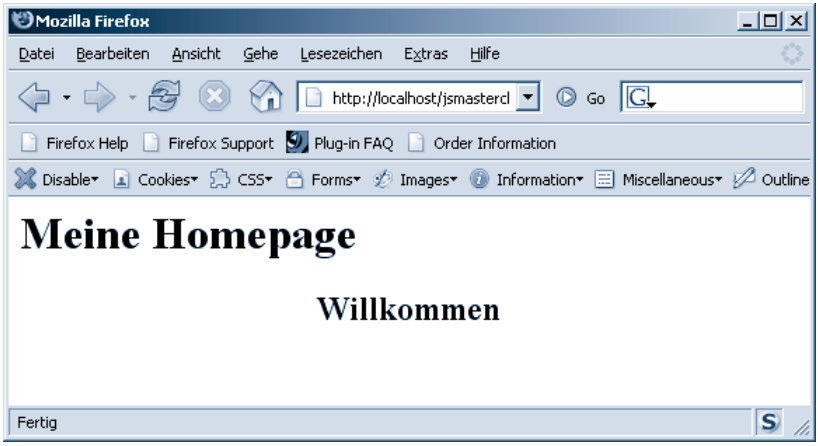


Abbildung 4.2
Teile der Webseite wurden dynamisch geschrieben.

Sie können wie schon angedeutet in einer Webseite mehrere verschiedene externe JavaScript-Dateien verwenden. Dabei sollten Sie aber beachten, dass es dann in den verschiedenen Dateien keine Funktionen oder Variablen mit gleichen Namen gibt. In dem Konfliktfall wird bei den meisten Browsern die zuletzt definierte Funktion oder Variable (im Sinn der Abarbeitung der Webseite von oben nach unten) verwendet. Aber darauf können Sie sich nicht verlassen und Sie sollten tunlichst eine solche Konfliktsituation vermeiden.

Hinweis

Sie müssen natürlich nicht alle in einer Datei oder einem Container deklarierten Funktionen auch wirklich nutzen.

4.3 Einbindung 3 – die Inline-Referenz

Der Aufruf einer JavaScript-Anweisung kann direkt in eine HTML-Referenz geschrieben werden. Dies ist die so genannte **Inline-Referenz**, die es in ähnlicher Form auch bei Style-Sheets gibt. Dieses System hat aber einige erhebliche Nachteile.

Die Schreibweise ist nicht gut lesbar (besonders bei mehreren Anweisungen, deshalb wird die Technik so gut wie ausschließlich zum Aufruf einer Anweisung oder Funktion verwendet) und weitgehend auf die Tags `<a>` und `<area>` eingeschränkt. Das heißt, es wird JavaScript-Code statt einer URL als Verweisziel definiert, indem dem Attribut `href` in Anführungszeichen eine oder mehrere JavaScript-Anweisungen zugewiesen werden. Dazu müssen Sie in einer entsprechenden HTML-Anweisung das Schlüsselwort `javascript` (als Protokoll zu verstehen), gefolgt von einem Doppelpunkt, als Attribut angeben. Die Syntax sieht also so aus:

Listing 4.9 Schema einer Inline-Referenz

```
<a href="javascript:[JavaScript-Anweisung]">
```

Beispiel:

Listing 4.10 Beispiel einer Inline-Referenz zum Aufruf bei einem Klick auf einen Hyperlink

```
<a href="javascript:alert('42')">Frage?</a>
```

Wenn der Anwender auf den Hyperlink klickt, wird mit `alert()` ein kleines Dialogfenster aufgeblendet, in dem der als Parameter angegebene Text angezeigt wird.

Schauen wir uns noch ein anderes, aber nun vollständiges Beispiel an (*inline-Referenz.html*):

Listing 4.11 Eine Webseite mit einer Inline-Referenz

```
01 <html>
02  <body>
03   <a href="javascript:location.href='http://www.ajax-net.de'">
04    AJAX-Portal</a>
05  </body>
06 </html>
```

In der Zeile 3 befindet sich die Inline-Referenz. In einem `<a>`-Element wird statt einem Hyperlink eine Weiterleitung per JavaScript mit `location.href` aufgerufen, wenn der Anwender den Link anklickt.

Achtung

Beachten Sie, dass Sie für den Fall, wo Sie Hochkommata innerhalb einer in Anführungszeichen eingeschlossenen Zeichenkette brauchen, nicht die normalen Hochkommata verwenden dürfen! Stattdessen können Sie das Zeichen `'` verwenden. Es entsteht also eine Verschachtelung mit zwei verschiedenen Zeichen. Etwa `"javascript:alert('Aber hallo')"` oder auch umgekehrt `'javascript:alert("Aber hallo")'`.

Hinweis

Die Verwendung der Inline-Referenz hat heute wirklich nur noch einen Exotenstatus. Sie ist nahezu vollständig durch die Verwendung des Eventhandlers `onClick` abgelöst worden. Allerdings gibt es ein Verhalten der Inline-Referenz, das von einigen Programmierern geschätzt wird – das Verändern des Mauszeigers zu einer Hand, wenn ein Anwender den Mauszeiger über den sensitiven Bereich bewegt. Der Anwender sieht also sofort, dass ein Klick an dieser Stelle eine Aktion auslöst. Wenn Sie in einen Hyperlink mit Inline-Referenz ein anderes HTML-Element wie eine Grafik oder eine Überschrift einschließen, können Sie ohne großen Aufwand dieses Mittel der Benutzerführung verwenden. Allerdings kann man den Mauszeiger auch mittels CSS und DHTML-Effekten verändern. Dies ist etwas aufwändiger, aber flexibler.

4.4 Der <noscript>-Container

Wenn Sie in einer Webseite ein Skript verwenden, ist es in einigen Situationen sinnvoll, dem `<script>`-Element nachfolgend einen `<noscript>`-Container zu notieren. In diesem können Informationen untergebracht werden, die Anwender zu sehen bekommen, bei denen JavaScript im Browser deaktiviert ist oder deren Browser nicht JavaScript-fähig[1] ist.

Im Wesentlichen bringt man in der Praxis darin Hinweise unter, dass zu einer vollständigen Funktionalität der Webseite JavaScript aktiviert sein muss, oder man stellt einen HTML-Hyperlink bereit, mit dem ein Anwender zu einem Webprojekt ohne die Verwendung von JavaScript gelangen kann. Browser mit (aktivierter) JavaScript-Fähigkeit werden den Inhalt des Containers ignorieren.

Eine Skriptreferenz mit `<noscript>`-Container sieht dann etwa so aus:

```
<script language="JavaScript">
<!--
  ...
//-->
</script>
<noscript>
Ihr Browser versteht entweder kein JavaScript
oder Sie haben es deaktiviert
<br />Für eine vollständige Funktionalität der
Webseite benötigen Sie jedoch JavaScript-Unterstützung.
</noscript>
```

Listing 4.12
Alternative Informationen können im <noscript>-Bereich notiert werden.

4.5 Angabe einer JavaScript-Version

JavaScript gibt es mittlerweile in mehreren Versionen, und natürlich sind gewisse Techniken erst sukzessive in bestimmten Versionen von JavaScript eingeführt worden. Wenn Sie also gewisse Techniken einsetzen, sollten Sie sicherstellen, dass sich auch nur Browser daran versuchen, die einen benötigten Sprachstandard unterstützen.

Zwar kann man mittlerweile »Web-weit« relativ unproblematisch die JavaScript-Version 1.3 voraussetzen, aber neuere Versionen – auch heutzutage – definitiv noch nicht.

Bei der Einbindung eines JavaScripts in eine Webseite können Sie als Ergänzung explizit eine JavaScript-Version angeben. Das erfolgt einfach, indem Sie beim Wert des `language`-Attributs ohne Leerzeichen (!) die Versionsnummer anfügen. Beispiel:

```
<script language="javascript1.3">
```

Listing 4.13
Angabe der JavaScript-Version 1.3

1. Diese Browser sollten aber wie gesagt ausgestorben sein.

Wenn Sie diese Angaben zur konkreten Version setzen, werden (bis auf wenige Ausnahmen – s. u.) in dem Container folgende Anweisungen nur von den Browsern ausgeführt, die JavaScript ab dieser Version auch unterstützen. Man kann damit die fehlerhafte Ausführung von Skripten durch inkompatible Browser verhindern, wobei die Container logisch sinnvoll aufgebaut werden müssen, um beim Ignorieren des Containers keine anderen Probleme zu bekommen.

Zumindest sollte es von der Theorie her so sein. In der Praxis gibt es jedoch einige potenzielle Fallen, weshalb dieses scheinbar banale Thema mehr Stoff hergibt, als es zuerst scheinen mag. So werden einige (sehr alte) Browser die Versionsanweisung nicht beachten und versuchen – trotz Inkompatibilität – diese für sie neuen und/oder für sie unbekannten Anweisungen auszuführen. Das Problem hat sich aber mittlerweile auf eine verschwindend kleine Anzahl an Browser-Methusalems reduziert.

Beim Internet Explorer muss man jedoch beachten, dass die Angabe des Tokens `JavaScript1.0` (was ja im Prinzip identisch zu der Angabe `JavaScript` ist) Probleme macht! Der Internet Explorer führt entsprechend gekennzeichnete Skriptcontainer einfach nicht aus. Man sollte deshalb **nie** `JavaScript1.0` für den `language`-Parameter angeben, sondern einfach nur das äquivalente `JavaScript`.

Allerdings können Sie den Internet Explorer mit der Angabe des MIME-Typs über `type="text/javascript"` dazu veranlassen, auch einen `<script>`-Container mit dem Attribut `language="JavaScript1.0"` auszuführen. Das hört sich im ersten Moment ganz sinnvoll an.

Aber das große Problem ist, dass der Internet Explorer bei der Angabe des MIME-Typs **alle** Versionsangaben vollkommen **ignoriert**. Das gilt leider auch für einige andere Browser! Ein Browser wird also auch Container mit Versionsangaben ausführen, die er unter Umständen nicht unterstützt.

Mit anderen Worten: Die **gleichzeitige** Angabe des MIME-Typs und einer Versionsangabe für JavaScript sollte unbedingt unterbleiben.

Hinweis

Probleme mit der Angabe der Version bei der gleichzeitigen Verwendung von `type` führen dazu, dass in der Praxis immer noch das vom W3C als deprecated erklärte `language`-Attribut Standard ist.

Das nachfolgende Beispiel kann zur Demonstration für so ein Fehlverhalten verwendet werden und allgemein ausgeben, welche Version von JavaScript sich ein Browser zutraut.

Beispiel (*versionsangaben.html*):

Listing 4.14
Eine Webseite mit mehreren Skriptcontainern für mehrere JavaScript-Versionen

```
<html>
 <body>
 <script language='JavaScript'>
 document.write("1. Jeder JavaScript-fähige Browser"
   + " - language='JavaScript'.<hr />");
 </script>
 <script language='JavaScript1.0'>
 document.write("2. Im Grunde identisch zu 1. "
   + " - language='JavaScript1.0'.<hr />");
 </script>
 <script language='JavaScript1.0' type='text/javascript'>
 document.write("3. Im Grunde identisch zu 1." +
   " - language='JavaScript1.0' type='text/javascript'.<hr />");
 </script>
 <script language='JavaScript1.1'>
 document.write("4. Alle Browser ab JavaScript 1.1"
   + " - language='JavaScript1.1'.<hr />");
 </script>
 <script language='JavaScript1.2'>
 document.write("5. Alle Browser ab JavaScript 1.2"
   + " - language='JavaScript1.2'.<hr />");
 </script>
 <script language='JavaScript1.3'>
 document.write("6. Alle Browser ab JavaScript 1.3"
   + " - language='JavaScript1.3'.<hr />");
 </script>
 <script language='JavaScript1.4'>
 document.write("7. Alle Browser ab JavaScript 1.4"
   + " - language='JavaScript1.4'.<hr />");
 </script>
 <script language='JavaScript1.5'>
 document.write("8. Alle Browser ab JavaScript 1.5"
   + " - language='JavaScript1.5'.<hr />");
 </script>
 <script language='JavaScript1.5' type='text/javascript'>
 document.write("9. Alle Browser ab JavaScript 1.5."
   + "<br />Aber über type='text/javascript' "
   + "werden einige Browser dies ausführen,"
   + " auch wenn die Version nicht implementiert ist.<hr />");
 </script>
 </body>
</html>
```

Wenn Sie die Datei in verschiedene Browser laden, werden Sie an der Ausgabe sehen, welche JavaScript-Versionen dieser nach eigener Einschätzung unterstützt. Beachten Sie, dass die Angabe des MIME-Typs im letzten Skriptcontainer nur das Problem mit der gleichzeitigen Angabe der Version und des MIME-Typs demonstrieren soll und keine Relevanz bezüglich der Version hat.

Abbildung 4.3
Der Internet Explorer 7 steigt ab Version 1.4 aus

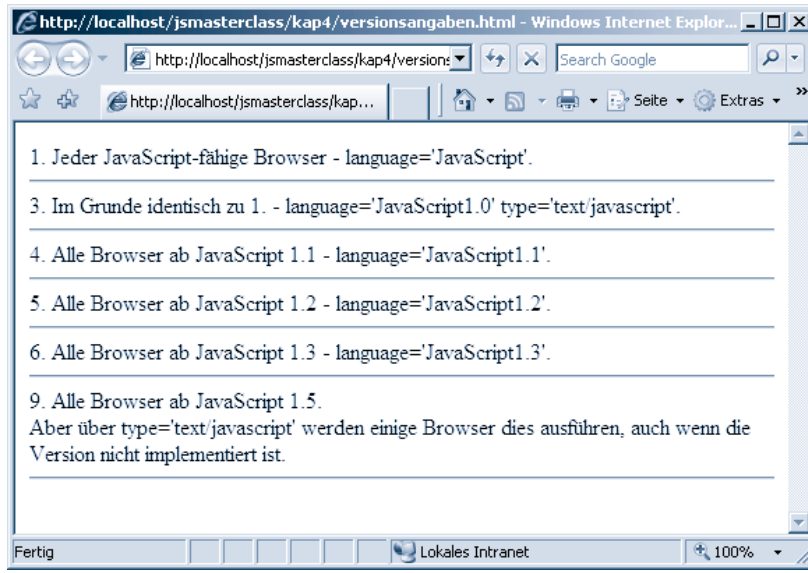
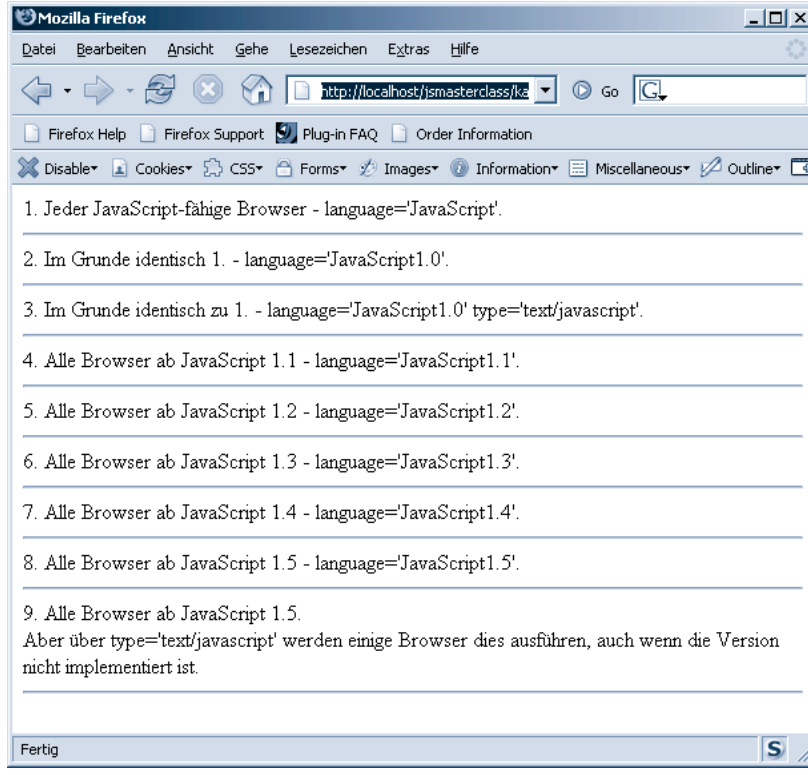


Abbildung 4.4
Der Firefox 1.5 kann alle Notationen.



Hinweis

Beachten Sie, dass Browser wie der Internet Explorer 6 oder 7 zwar Versionsangaben jenseits von `JavaScript1.3` als Bremse betrachten und in dem Skriptcontainer notierte Anweisungen nicht ausführen. Das bedeutet allerdings nicht, dass solche Browser Anweisungen späterer Sprachversionen überhaupt nicht können. Der Internet Explorer 6 und erst recht 7 beherrscht z. B. alle relevanten Anweisungen der Version JavaScript 1.5. Er wird nur an den Token `JavaScript1.4` und `JavaScript1.5` scheitern.

4.6 Wie kann ich sicherstellen, dass ein Browser nur solche JavaScript-Anweisungen ausführt, die er versteht?

Diese Anleitung mag im ersten Moment überflüssig erscheinen, denn auf den ersten Blick sieht es so aus, als würde es genau der Angabe einer JavaScript-Version entsprechen. Die Angabe der Version erweist sich jedoch wie gesagt in vielen Fällen schlicht und einfach als unbrauchbar, weil sie zu restriktiv ist. Es gibt viele Anweisungen, die offiziell zu einer bestimmten Version von JavaScript gehören und die dennoch von Browsern verstanden werden, die einen bestimmten Standard nicht vollständig unterstützen und deshalb einen explizit mit einer Versionsangabe gekennzeichneten Container vollständig ignorieren werden. Das gilt vor allem für Techniken aus dem JavaScript-Standard 1.5.

Beispielsweise ignoriert, wie gesehen, der Internet Explorer 7 alle Container, die mit Versionsangaben jenseits von `javascript1.3` gekennzeichnet sind. Dabei unterstützt der Internet Explorer 6 den Standard 1.5 nahezu vollständig. Ebenso wird beispielsweise Opera 7.2 nur Container bis zur Angabe `javascript1.4` ausführen, obwohl JavaScript 1.4 im Grunde nie ein implementierter Standard war und auch dieser Browser nahezu mit allen Anweisungen aus JavaScript 1.5 zurechtkommt (bereits in Vorgängerversionen). Nachfolgend finden Sie in einer Tabelle, welche wichtigen Browserversionen offiziell mit welcher Version von JavaScript mindestens zurechtkommen[1]:

	JavaScript-Version					
Browser	1.0	1.1	1.2	1.3	1.4	1.5
Netscape Navigator 2.x	x					
Netscape Navigator 3.x	x	x				
Netscape Navigator bzw. Communicator 4.0	x	x	x			

Tabelle 4.1
Die offizielle Kompatibilität wichtiger Browserversionen mit JavaScript-Standards

1. Wie gesagt – einige Browser können auch Techniken späterer Versionen verstehen.

Tabelle 4.1 (Forts) Die offizielle Kompatibilität wichtiger Browserversionen mit JavaScript-Standards

	JavaScript-Version					
Browser	1.0	1.1	1.2	1.3	1.4	1.5
Netscape Navigator bzw. Communicator > 4.05 bis 4.7	x	x	x	x		
Netscape Navigator bzw. Communicator 6.x	x	x	x	x	x	x
Netscape Navigator 7.x ff	x	x	x	x	x	x
Mozilla 1.2	x	x	x	x	x	x
Mozilla 1.4 ff.	x	x	x	x	x	x
Firefox	x	x	x	x	x	x
Opera 5.x	x	x	x	x		
Opera 6.x	x	x	x	x	x	
Opera 7.x	x	x	x	x	x	
Opera 8 ff	x	x	x	x	x	x
Microsoft Internet Explorer 2.x						
Microsoft Internet Explorer 3.x	x					
Microsoft Internet Explorer 4.x	x	x	x			
Microsoft Internet Explorer 5.x	x	x	x	x		
Microsoft Internet Explorer 6.x	x	x	x	x		
Microsoft Internet Explorer 7.x	x	x	x	x		

Für die Praxis kann man festhalten, dass ein entsprechender Container mit Versionsangabe bis zu Anweisungen der JavaScript-Version 1.3 (inklusive) Sinn macht. Darüber hinaus sollte man bei Bedarf die Browservariante und die Browserversion abfragen. Darauf kommen wir im Laufe des Buches noch. Universell können Sie also folgende Schablone einsetzen, die allerdings mit den Angaben sehr früher JavaScript-Versionen sehr konservativ gestaltet ist (*versionsschablone.js*):

Listing 4.15 Eine Schablone zum qualifizierten Sicherstellen der Unterstützung bestimmter Anweisungen

```
<script language='JavaScript'>
 <!--
 /* Notieren Sie hier alle JavaScript-Anweisungen, die alle Java-
Script-fähigen Browser verstehen */
 ...
 //-->
 </script>
 <script language='JavaScript1.1'>
 <!--
 /* Notieren Sie hier alle JavaScript-Anweisungen, die Browser
voraussetzen, die mindestens JavaScript 1.1 unterstützen */
 ...
 //-->
```

```
 </script>
 <script language='JavaScript1.2'>
 <!--
 /* Notieren Sie hier alle JavaScript-Anweisungen, die Browser
voraussetzen, die mindestens JavaScript 1.2 unterstützen */
 ...
 //-->
 </script>
 <script language='JavaScript1.3'>
 <!--
 /* Notieren Sie hier alle JavaScript-Anweisungen, die Browser
voraussetzen, die mindestens JavaScript 1.3 unterstützen */
 ...
 /* Bauen Sie im Inneren dieses Containers für kritische Anweisungen
eine qualifizierte Browserweiche für die Browser auf, die mit diesen
Anweisungen klarkommen, offiziell jedoch nur JavaScript 1.3 unter-
stützen */
 //-->
 </script>
```

Listing 4.15 (Forts.) Eine Schablone zum qualifizierten Sicherstellen der Unterstützung bestimmter Anweisungen

4.7 Kann man testen, ob bei einem Browser JavaScript aktiviert ist?

JavaScript wird in allen relevanten modernen Webclients unterstützt. Dennoch sollten Sie sich nicht darauf verlassen, dass sie bei einem Besucher unterstützt wird. Ein auf Sicherheit bedachtes Webprojekt sollte niemals JavaScript so verwenden, dass es ohne die Unterstützung von JavaScript vollkommen unbrauchbar ist.

Browser ohne JavaScript-Unterstützung können Sie mittlerweile vernachlässigen, aber viele Anwender deaktivieren JavaScript. Sei es aus Unkenntnis, weil sie eine Gefahr befürchten, weil es entsprechende Unternehmensrichtlinien gibt oder auch, weil JavaScript oft zur Programmierung von nervigen Dingen wie Werbe-Popups verwendet wird.

Ein kurzer Test auf die Unterstützung von JavaScript ist also unabdingbar. Dieser Test kann entweder ein einziges Mal beim Einstieg in ein Webprojekt ausgeführt werden[1] oder auch bei jedem Laden einer Webseite Ihres Projekts[2]. Die explizite Abfrage bei jedem Laden einer neuen Webseite beugt dem Anwenderverhalten vor, dass während eines Besuchs auf Ihrem Webprojekt vom Anwender die JavaScript-Unterstützung erst deaktiviert wird und der Test auf der Einstiegsseite JavaScript als aktiviert gekennzeichnet hatte. Wenn Sie das nicht tolerieren können, können Sie zumindest beim Seitenwechsel gegensteuern.

1. Also in der Regel beim Laden der Seite *index.html*, *index.php* etc., die bei den meisten Webprojekten als Einstiegsseite verwendet werden – das sollte in der Regel genügen.
2. Was meines Erachtens aber überzogen ist.

Wie können Sie nun testen, ob bei einem Client JavaScripts überhaupt ausgeführt werden können oder nicht? Mit JavaScript selbst? Eine Abfrage per JavaScript, ob JavaScript aktiviert ist, ist ja nicht möglich, wenn es deaktiviert ist. Oder doch? Es geht wirklich. Indirekt. Man nutzt explizit die Tatsache, dass eine JavaScript-Aktion bei deaktiviertem JavaScript überhaupt nicht funktioniert.

Das geht beispielsweise so: Mit JavaScript führen Sie eine **Weiterleitung** zu einem anderen URL aus. Funktioniert der, muss auch JavaScript aktiviert sein. Wenn Sie den Anwender nun komfortabel führen wollen, erstellen Sie eine Einstiegsseite in Ihr Webprojekt, das im Wesentlichen eine solche Weiterleitung enthält und ihn für den Fall vom deaktivierten JavaScript automatisch zu einem Projekt führt, das ohne JavaScript auskommt.

Wie wir im letzten Kapitel gesehen haben, funktioniert eine solche Weiterleitung auch ohne JavaScript. Mit HTML sind automatische Weiterleitungen mit dem Tag `<meta http-equiv="refresh" content="[Zeit];URL=[Ziel]">` möglich. Beim Attribut `content` geben Sie zuerst die Zeitspanne an, nach der die Weiterleitung erfolgen soll, und dann das Ziel der Weiterleitung. Der entscheidende Trick beruht nur darauf, die Weiterleitung per HTML erst **nach** der Weiterleitung per JavaScript auszuführen. Das nachfolgende Beispiel zeigt so eine automatische Trennung nach JavaScript und deaktiviertem JavaScript.

Beispiel (*weiterleitung.html*):

Listing 4.16
Trennung nach JavaScript-Unterstützung oder nicht

```
01 <html>
02 <head>
03   <meta http-equiv="refresh" content="8;URL=ohneJS.html">
04 </head>
05 <script language="JavaScript">
06   location.href="mitJS.html";
07 </script>
08 <body>
09 <a href="ohneJS.html">Wenn die automatische Weiterleitung
10 nicht funktioniert, klicken Sie bitte hier, um zum Webprojekt
11 ohne JavaScript-Unterst&uuml;tzung zu kommen</a><br />
12 <a href="mitJS.html">Wenn die automatische Weiterleitung
13 nicht funktioniert, klicken Sie bitte hier, um zum Webprojekt
14 mit JavaScript-Unterst&uuml;tzung zu kommen</a>
15 </body>
16 </html>
```

Wenn man diese Webseite lädt, erfolgt eine automatische Weiterleitung per JavaScript mit dem `location`-Objekt (`location.href` – Zeile 6). Nun wird diese Anweisung bei deaktiviertem JavaScript nicht ausgeführt. Im Header finden Sie in Zeile 3 einen HTML-`<meta>`-Tag, der aber erst nach einem Zeitintervall (hier acht Sekunden und im Allgemeinen unbedingt länger als das Weiterleitungsintervall des JavaScripts) automatisch auf ein Teilprojekt ohne JavaScript weiterleitet. Zur Sicherheit notieren Sie noch zwei Hyperlinks, womit die Weiterleitung auch vom Besucher manuell ausgelöst werden kann.

Achtung

Sie können in jedem Fall davon ausgehen, dass JavaScript aktiviert ist, wenn die Weiterleitung oder auch das Öffnen eines neuen Fensters funktioniert hat. Der Umkehrschluss ist jedoch nicht zwingend. In einigen neueren Browsern können Anwender sehr differenziert angeben, welche JavaScript-Aktionen sie tolerieren.

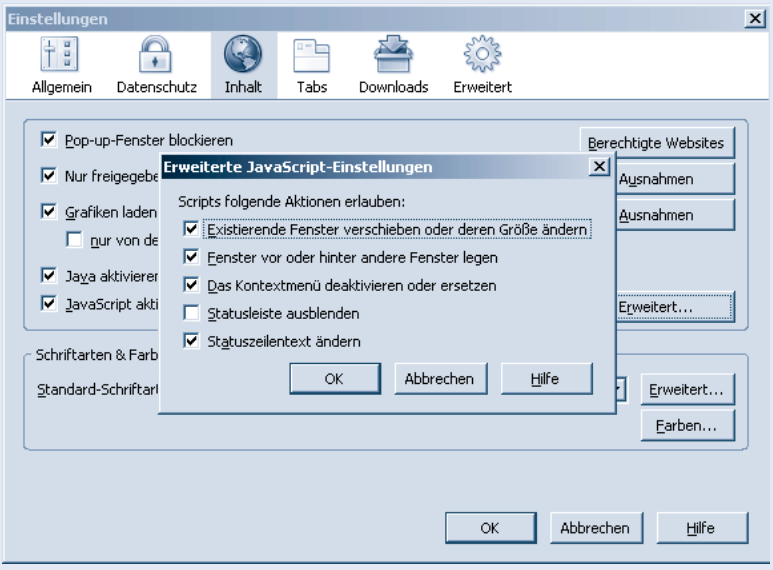


Abbildung 4.5
Differenzierte Einstellungen, was mit JavaScript erlaubt ist – hier Firefox

4.8 Kann man testen, welche JavaScript-Version von einem Browser unterstützt wird?

Direkt können Sie die JavaScript-Version, die im Browser eines Besuchers unterstützt wird, nicht abfragen, aber indirekt. Wenn Sie beim `<script>`-Container explizit eine Version für JavaScript angeben und eine Folge von `<script>`-Containern intelligent kombinieren, können Sie indirekt und sehr zuverlässig die unterstützte JavaScript-Version von einem Browser testen. Das funktioniert zumindest bis zur Version 1.3 sehr verlässlich und mit Einschränkungen auch für neuere Versionen[1].

1. Darüber hinaus sind keine allgemeinen Aussagen.

Beispiel (*versionsunterstuetzung.html*):

Listing 4.17 Ein indirekter Test, welche Version von JavaScript unterstützt wird

```
<html>
<body>
<script language="JavaScript1.5">
  location.href="index1_5.html";
</script>
<script language="JavaScript1.4"> // im Grunde überflüssig
  location.href="index1_4.html";
</script>
<script language="JavaScript1.3">
  location.href="index1_3.html";
</script>
<script language="JavaScript1.2">
  location.href="index1_2.html";
</script>
<script language="JavaScript1.1">
  location.href="index1_1.html";
</script>
<script language="JavaScript">
  location.href="index1_0.html";
</script>
</body>
</html>
```

Der eigentliche Trick basiert darauf, dass Sie die Container in umgekehrter Reihenfolge notieren (zuerst die Version 1.5, dann 1.4, 1.3 bis 1.0). Im Inneren der jeweiligen Container verwenden Sie im Grunde den gleichen Trick wie beim Test, ob JavaScript aktiviert ist. Sie leiten automatisch auf eine Folgeseite weiter, die zu einer bestimmten JavaScript-Version passt und für dieses Projekt als Einstiegsseite dient. Beim ersten Treffer wird zu der angegebenen Seite verzweigt und die Abarbeitung von dem Testskript abgebrochen. Damit ist sichergestellt, dass die entsprechende JavaScript-Version unterstützt wird.

4.9 Übungsaufgaben

Aufgabe 1 Testen Sie die verschiedenen Möglichkeiten zum Einbinden von JavaScripts.

Aufgabe 2 Analysieren Sie die Quelltexte von verschiedenen Webseiten und versuchen Sie Referenzen auf JavaScript-Dateien, interne JavaScript-Container und Inline-Referenzen zu finden. Sie werden bei fast allen populären Webangeboten fündig werden.

Aufgabe 3 Betrachten Sie noch einmal die JavaScript-Beispiele von Kapitel 2. Merken Sie sich bereits interessante JavaScript-Möglichkeiten, wie das Weiterleiten per `location.href`, das dynamische Schreiben von Teilen der Webseite mit `document.write()`, das Anzeigen eines Auswahldialogs mit `confirm()` oder das Anzeigen einer Mitteilungsbox mit `alert()`.