PEARSON Education

Markt+Technik

Ralph Steyer

JavaScript

- Scripte einbinden und programmieren
- Alles zu Stylesheets, DHTML, Objekten und Formularen

Markt + Technik Verlag

Scripte in Webseiten einbinden

Gestern haben Sie wichtige Grundlagen zum Erstellen einer Webseite kennen gelernt. Dazu zählen der Aufbau von Steueranweisungen in HTML/XHTML und das Grundgerüst einer Webseite. Heute kommen wir zur Integration von JavaScript in Webseiten. Im Wesentlichen gibt es folgende drei Möglichkeiten (es gibt noch mehr, aber abgesehen von einer Speziallösung für den Internet Explorer gehen wir auf diese nicht weiter ein):

- Die Quelltexte werden direkt in einen HTML-Container in einer Webseite notiert.
- Eine externe Datei wird mit der Webseite verknüpft.
- Zur Einbindung wird eine Erweiterung eines HTML-Tags verwendet.

Dieser Tag wird zwar nicht sonderlich umfangreich, ist aber dennoch von grundlegender Bedeutung.

3.1 Scripte in Webseiten

Im Prinzip ist es schon sehr lange möglich, in HTML Scripte zu integrieren. Aber erst seit HTML 3.2 können Scripte über standardisierte Wege in Webseiten eingebunden werden. Sie werden dazu entweder über genormte Schritte als Klartext direkt in HTML-Seiten notiert oder dort auf normierte Weise über eine externe Datei referenziert. Dazu gibt es im Wesentlichen den standardisierten HTML-Tag `<script>`, der den Beginn eines Script-Containers definiert. Das Innere des Containers wird gegenüber dem HTML-Interpreter geschützt, während der Tag selbst samt seiner Attribute noch Bestandteil des HTML-Codes ist[1]. Im Inneren des Containers aktiviert ein scriptfähiger Browser seinen Script-Interpreter.

Man kann sich das so vorstellen (ein Vergleich mit einer Firma): Wenn eine Webseite geladen wird, werden die jeweiligen Zeichen der Reihe nach von oben nach unten von der HTML-Abteilung bearbeitet, bis der `<script>`-Tag entdeckt wird. Daraufhin wird der Fall an die zuständige Script-Abteilung weitergeleitet, die den Vorgang so lange bearbeitet, bis das Ende des `</script>`-Containers erreicht wird. Anschließend übergibt die Script-Abteilung den Vorgang wieder an die HTML-Abteilung, welche den Vorgang weiterbearbeitet.

Im Zusammenhang mit der Analyse der Webseite fällt oft der Begriff Parser. Unter einem Parser versteht man ein Programm oder Programmmodul, das die Strukturen einer Beschreibungs-/Script-/Programmiersprache erkennen kann.

Eine Script-Referenz kann im Prinzip an jeder Stelle in der Webseite notiert werden, solange sie innerhalb des äußersten HTML-Containers – dem vom `<html>`-Tag gebildeten Block – steht. Am Beginn des Body-Teils der Seite, am Ende des Body, mitten drin oder gar

1 Deshalb spielt hier Groß- und Kleinschreibung in der Praxis keine Rolle. Aber mit dem schon mehrfach erwähnten Hinweis auf XML bzw. XHTML sollte man Kleinschreibung verwenden.

zwischen Header und Body (obwohl davon von einigen Experten abgeraten wird). Oft wird sie im Header der Seite oder direkt dahinter zu finden sein. Das hat mehrere Gründe. Einmal ist es in Hinblick auf die Wartung leichter, wenn man die Informationen zusammen verwaltet, die logisch zusammengehören. Logisch gehören Scripte nicht zu den darstellbaren HTML-Elementen. Deshalb werden bei einer Platzierung im Header oder direkt dahinter Layout und Hintergrundaktionen besser getrennt. Nebenbei wird das Auffinden von Scripten erleichtert, wenn Code immer zentral außerhalb des meist recht umfangreichen Bodys notiert wird. Das entscheidende Argument ist jedoch der Ladevorgang für eine Webseite – von oben nach unten.

Um ein wenig vorzugreifen – Script-Schritte können einmal einfach direkt in den Script-Container notiert werden. Solcher Code wird ausgeführt, wenn die entsprechenden Anweisungen beim Laden der Webseite vorgefunden werden. Der Code kann damit auch nur genau einmal ausgeführt werden. Anwendung für eine solche Vorgehensweise sind hauptsächlich Initialisierungen oder das dynamische Generieren von statischen Webseiten.

Der andere Weg zur Notation von Scripten ist der, dass Anweisungen über Funktionen zusammengefasst und dann gemeinsam über einen Namen aufgerufen werden (es müssen nicht unbedingt mehrere Anweisungen zusammengefasst werden – es kann auch nur eine Anweisung in einer Funktion notiert werden). Ein solcher als Funktion gekennzeichneter Code wird erst dann ausgeführt, wenn an irgendeiner anderen Stelle im HTML-Code der Name der Funktion aufgerufen wird. Funktionen sind im gesamten Dokument verfügbar und können bei Bedarf immer wieder aufgerufen werden.

Wenn nun eine JavaScript-Funktion beim Laden einer Webseite direkt am Anfang des Bodys benötigt und sie dort aufgerufen wird, deren konkrete Programmierung aber erst am Ende der Webseite zu finden ist, kann es zu Problemen kommen, besonders im Internet, wo die Übertragung einer Webseite immer wieder ins Stocken geraten kann. Deshalb notiert man die Programmierung einer Funktion besser vor deren erst möglichen Aufruf.

3.2 Die Verwendung des <script>-Containers

Der `<script>`-Tag kann zur Kennzeichnung von Script-Bereichen ohne irgendwelche Attribute verwendet werden, etwa so:

```
<script>
      ... irgendwelche Script-Anweisungen
</script>
```

In diesem Fall wird der HTML-Interpreter beim Antreffen des Anfang-Tags `<script>` die Verantwortung an den Default-Script-Interpreter des Browsers abgeben. In der Regel ist das ein JavaScript-Interpreter, worauf man sich aber nicht verlassen kann. Man sollte sich immer vergegenwärtigen, dass ein Browser verschiedenste Script-Interpreter aufrufen kann, sofern sie ihm zur Verfügung stehen.

Um die verwendete Script-Sprache explizit anzugeben, wird der `<script>`-Tag um das Attribut `language` (Sprache) erweitert, welches bestimmt, um welche Script-Sprache es sich in dem nachfolgenden Container handelt. Die Syntax `<script language="JavaScript">` informiert den Browser, dass innerhalb des Containers ein JavaScript folgt. In einigen Browserversionen kann sogar der ursprüngliche Name von JavaScript – `LiveScript` – stehen. Dies ist aber selten sinnvoll und wird nicht von allen Browsern unterstützt.

Die letzten Ausführungen sind ebenfalls für die Angabe von `JScript` zu sehen. Für die Ausführung von Scripten im Internet Explorer kann man die Syntax `<script language="JScript">` angeben, wenn explizit die Microsoft-JavaScript-Variante verwendet werden soll. Dann aber grenzt man die Anwender anderer Browser aus, was selten notwendig oder gewünscht ist. Man kann allerdings dieses Verhalten nutzen, um Browserwelten zu trennen, was gelegentlich wegen der Inkompatibilitäten in Scripten sinnvoll ist. Wir werden darauf zurückkommen.

Man ist bei der Angabe einer Script-Sprache also nicht auf JavaScript oder Abarten davon beschränkt. Jede im Browser unterstützte Script-Sprache lässt sich nach dem gleichen Schema referenzieren. Die Angabe `<script language="VBScript">` legt beispielsweise VBScript als Script-Sprache im Container fest. Die konkrete Syntax für den Script-Container sieht also schematisch immer so aus:

```
<script language="[Sprache]" type=[MIME-Typ]>
      ... irgendwelche Script-Anweisungen
</script>
```

Der `<script>`-Container beinhaltet also bei einer Referenzierung von JavaScript im Inneren JavaScript-Anweisungen, mit denen ein HTML-Interpreter nichts anfangen kann. Wenn ein Browser JavaScript versteht, wird von diesem deshalb der Vorgang an den JavaScript-Interpreter weitergegeben, aber nicht-JavaScript-fähige Browser haben mit dem im Script-Container notierten Code Probleme. Zwar wird der `<script>`-Tag samt Abschluss-Tag aufgrund des Prinzips der Fehlertoleranz von diesen ignoriert. Da er jedoch nicht verstanden wird, werden die darin notierten Script-Befehle als HTML-Code interpretiert (der Browser weiß ja nichts von Scripten). In der Regel heißt das, die Script-Anweisungen werden als Klartext verstanden, der einfach in den Anzeigebereich des Browsers geschrieben wird, was so gut wie nie wünschenswert ist.

Deshalb steht normalerweise als Vorsichtsmaßnahme bei einem Script-Container direkt nach dem einleitenden <script>-Tag ein öffnendes HTML-Kommentarzeichen (<!--) – siehe dazu Tag 2. Dieser HTML-Kommentarcontainer wird unmittelbar vor dem abschließenden Tag </script> mit dem entsprechenden HTML-Kommentarzeichen (-->) wieder geschlossen. Dadurch steht der gesamte Script-Code innerhalb eines HTML-Kommentars. Dies ist, wie gesagt, nicht zwingend, aber sicherer für den Fall, dass ein Browser, der keine JavaScripte interpretieren kann, die Seite lädt.

Nun gibt es jedoch einige Browser (etwa einige Versionen vom Netscape Navigator), die beim Einschluss von Scripten in HTML-Kommentare ziemlich gewöhnungsbedürftig reagieren. Der innerhalb des <script>-Containers aktive Script-Interpreter kann mit der Zeichenkombination für das Ende eines HTML-Kommentars nichts anfangen. Genau genommen erkennt er nicht, dass es eine HTML-Anweisung ist, sondern interpretiert -- als JavaScript-Operator (diesen Operator gibt es) und erzeugt damit einen Fehler. Deshalb wird der Ende-Tag des HTML-Kommentars vor dem Script-Interpreter mit dem JavaScript-Kommentarzeichen // versteckt (das lernen wir noch genauer kennen). Die Anweisung vor dem Ende des <script>-Containers sieht dann so aus: //-->.

Die ganze Problematik betrifft nicht die Browser, in denen JavaScript nur deaktiviert ist. Diese werden den <script>-Container schon erkennen, nur die darin enthaltenen Anweisungen nicht ausführen. Aber das ist in gewisser Weise auch ein Problem, denn ein Besucher kann nicht erkennen, warum eine Webseite nicht so funktioniert, wie man es erwarten kann (oder noch schlimmer – gar nichts anzeigt).

Für die JavaScript-unfähigen Browser (inklusive der Browser, wo der Anwender JavaScript deaktiviert hat) sollte man normalerweise dem <script>-Container einen weiteren <noscript>-Container folgen lassen, der so aussieht:

```
<noscript>Sorry, diese Seite verwendet JavaScript!</noscript>
```

Der Text (der natürlich frei wählbar ist) wird dabei explizit nicht in HTML-Kommentare eingeschlossen. Hier ist die Situation ja eine andere. JavaScript-fähige Browser werden den Tag erkennen und den darin enthaltenen Inhalt ignorieren. JavaScript-unfähige Browser werden diesen Tag genauso wenig kennen wie den <script>-Tag. Das ist aber genauso sinnvoll. Dies bedeutet doch, dass er ignoriert und nachfolgender Text einfach in der Webseite angezeigt wird. Bingo! So soll es ja auch sein.

Das auch gegen JavaScript-unfähige Browser und solche mit deaktivierter JavaScript-Einstellung geschützte, vollständige Grundgerüst sieht also dann so aus:

```
<script language="JavaScript" type="text/javascript">
<!--
      ... irgendwelche Script-Anweisungen
// -->
</script>
<noscript>Sorry, diese Seite verwendet JavaScript!</noscript>
```

Eine JavaScript-Einbindung mit vollständigem Grundgerüst

Führen wir nun ein kleines Beispiel durch, das ein JavaScript mit vollständigem Grundgerüst einbindet. Die Anweisung wird direkt beim Laden der Webseite ausgeführt. Geben Sie im Editor den nachfolgenden Quelltext ein:

Listing 3.1: Script-Einbindung mit vollständigem <script>-Container

```
01 <html>
02   <body>
03   <script language="JavaScript" type="text/javascript">
04   <!--
05         document.write(
             "Diese Webseite befindet sich an folgendem URL: " +
             document.location);
06   // -->
07   </script>
08   <noscript>Sorry, diese Seite verwendet JavaScript!</noscript>
09 </body>
10 </html>
```

Speichern Sie den Text und schauen Sie sich die Datei im Browser an. Die JavaScript-Anweisung in Zeile 5 gibt Ihnen den URL an, von dem die Seite geladen wurde.

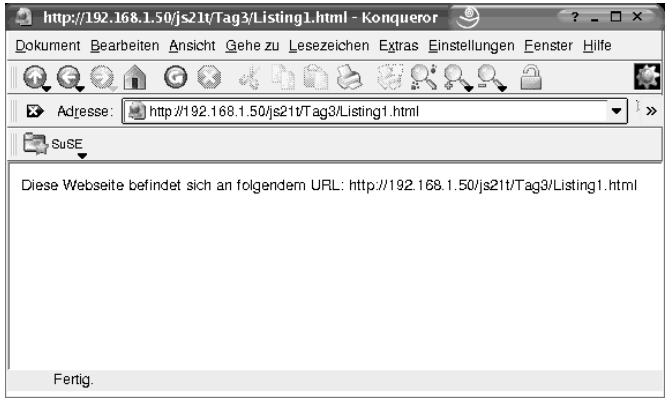

Abbildung 3.1: Mit aktiviertem JavaScript bekommen Sie den URL angezeigt, von dem Sie die Datei geladen haben.

Deaktivieren Sie nun JavaScript und laden Sie die Datei erneut in den Browser. Sie bekommen den Inhalt des `<noscript>`-Containers angezeigt. Sollte Ihnen noch ein alter Browser ganz ohne JavaScript-Unterstützung zur Verfügung stehen[2], testen Sie die Datei auch darin.

Abbildung 3.2: Der Inhalt des <noscript>-Containers – hier im Konqueror 3

Wir werden in der Folge sowohl auf die Angabe des MIME-Typs als auch auf den <noscript>-Container verzichten.

Die Anzahl der Script-Container in einer Webseite ist nicht beschränkt. Es lassen sich mehrere <script>-Container in einem Dokument platzieren. Es gibt ein paar sinnvolle Anwendungen, warum nicht alle Scripte in einem Container an einer Stelle notiert werden. Wichtigster Grund ist die Verwendung von verschiedenen Script-Sprachen oder -Versionen innerhalb einer Webseite. Die Aufteilung in verschiedene Container erlaubt eine qualifizierte Auswahl, welche Anweisungen von welchen Browsern ausgeführt werden. Oder man möchte mögliche Funktionalitäten aus verschiedenen Quellen in einer Webseite nutzen. Eine weitere Anwendung für verschiedene <script>-Container ist die damit mögliche logische Strukturierung von Inhalten.

Die Angabe der Version von JavaScript

Was gelegentlich sinnvoll ist, ist die Angabe einer konkreten Version von JavaScript. Dies ist etwa dann zweckmäßig, um Fehler mit inkompatiblen Browsern vorzubeugen. Wenn einfach nur die Sprache allgemein spezifiziert wird, wird jeder Browser, der diese Sprache versteht, das Script abarbeiten. Sofern dann dort Befehle aus einer Sprachversion genutzt werden, die der Browser nicht unterstützt, kann es zu Komplikationen kommen, etwa bei der Kombination von JavaScript-Befehlen der Version 1.2 und dem Browser Internet Explorer 3.0 oder JavaScript-Befehlen der Version 1.3 und dem Netscape Navigator 3. Über die explizite Versionsangabe kann verhindert werden, dass die betroffenen inkompatiblen Browser diese Scripte ausführen. Bei JavaScript werden die Versionen wie folgt angegeben:

2 Vielleicht arbeiten Sie ja in einem Museum oder haben kurz nach Ablösung der Trommeln durch das Telefon einen Rechner samt Software auf dem Speicher eingemottet ;-).

- `<script language ="JavaScript1.0">`
- `<script language ="JavaScript1.1">`
- `<script language ="JavaScript1.2">`
- `<script language ="JavaScript1.3">`
- `<script language ="JavaScript1.4">`
- `<script language ="JavaScript1.5">`

Wenn man das Verhalten mit verschiedenen Script-Containern für verschiedenartige Browser koppelt, kann man sehr qualifiziert differenzieren. Wir werden im Rahmen unserer Beispiele auf die explizite Angabe der JavaScript-Version verzichten, aber ein Beispiel zum Differenzieren von Browsern und der JavaScript-Version zeigen.

Die Versionsnummer muss ohne Leerzeichen angefügt werden! Es spielt jedoch keine Rolle, ob `Javascript` oder `JavaScript` (meist gewählte Variante) oder `javascript` als Wert genommen wird, d.h. Groß- und Kleinschreibung spielen hier keine Rolle. Das ist klar, wenn man sich vergegenwärtigt, dass diese Anweisungen noch zu HTML zählen. Erst in den konkreten, im Container notierten Anweisungen wird Groß- und Kleinschreibung eine Rolle spielen – dort gilt JavaScript-Syntax.

Beim Internet Explorer muss man beachten, dass die Angabe `JavaScript1.0` (was ja identisch mit `JavaScript` ist) Probleme bereitet (siehe Listing 3.2 und Abbildung 3.4). Man sollte deshalb nie `JavaScript1.0` für den `language`-Parameter angeben, sondern einfach nur das äquivalente `JavaScript`.

Eine Webseite mit mehreren Script-Containern und Versionsangaben

Wir wollen kurz eine Webseite erstellen, in der mehrere Script-Container mit verschiedenen JavaScript-Versionsangaben notiert sind.

Listing 3.2: Eine Webseite mit mehreren Script-Containern für mehrere JavaScript-Versionen

```
01 <html>
02   <body>
03     <script language="JavaScript">
04     <!--
05       document.write("Das schreibt jeder Browser, der JavaScript kann.<br>");
06     //-->
```

```
07    </script>
08     <script language="JavaScript1.0">
09     <!--
10       document.write("Das ist identisch mit oben - es schreibt jeder Browser,
         der JavaScript kann.<br>");
11     //-->
12    </script>
13    <h2>Zur Abwechselung reines HTML</h2>
14     <script language="JavaScript1.1">
15     <!--
16       document.write("Das können nur Browser, die JavaScript ab Version 1.1
         können.<br>");
17     //-->
18    </script>
19     <script language="JavaScript1.2">
20     <!--
21       document.write("Das können nur Browser, die JavaScript ab Version 1.2
         können.<br>");
22     //-->
23    </script>
24     <script language="JavaScript1.3">
25     <!--
26       document.write("Das können nur Browser, die JavaScript ab Version 1.3
         können.<br>");
27     //-->
28    </script>
29     <script language="JavaScript1.4">
30     <!--
31       document.write("Das können nur Browser, die JavaScript ab Version 1.4
         können.<br>");
32     //-->
33    </script>
34     <script language="JavaScript1.5">
35     <!--
36       document.write("Das können nur Browser, die JavaScript ab Version 1.5
         können.");
37     //-->
38    </script>
39   </body>
40 </html>
```

Beachten Sie, dass zwischen den ganzen Ausgaben per JavaScript eine reine HTML-Anweisung steht (Zeile 13). In der Ausgabe beim Besucher ist nicht zu unterscheiden, ob ein Teil der Webseite statisches HTML oder eine dynamisch generierte Information war. Beachten Sie auch, dass in der JavaScript-Ausgabe neben reinem Text auch HTML-Anweisungen (`<br>`) geschrieben werden.

Laden Sie die Datei in einen alten Browser, der etwa nur JavaScript 1.1 kann, werden Sie die folgende Ausgabe sehen:

```
Das schreibt jeder Browser, der JavaScript kann.
Das ist identisch mit oben - es schreibt jeder Browser, der JavaScript kann.
Zur Abwechselung reines HTML
Das können nur Browser, die JavaScript ab Version 1.1 können.
```

Kann der Browser jedoch JavaScript 1.3 (was derzeit alle aktuellen Browser verstehen), sehen Sie Folgendes:

```
Das schreibt jeder Browser, der JavaScript kann.
Das ist identisch mit oben - es schreibt jeder Browser, der JavaScript kann.
Zur Abwechselung reines HTML
Das können nur Browser, die JavaScript ab Version 1.1 können.
Das können nur Browser, die JavaScript ab Version 1.2 können.
Das können nur Browser, die JavaScript ab Version 1.3 können.
```

Abbildung 3.3:
Im Opera 6 sehen Sie alle Ausgaben der JavaScript-Anweisungen bis inklusive 1.4.

Eine Ausnahme ist der Internet Explorer, der – wie schon oben bemerkt – mit der Angabe *Javascript1.0* seine Probleme hat.

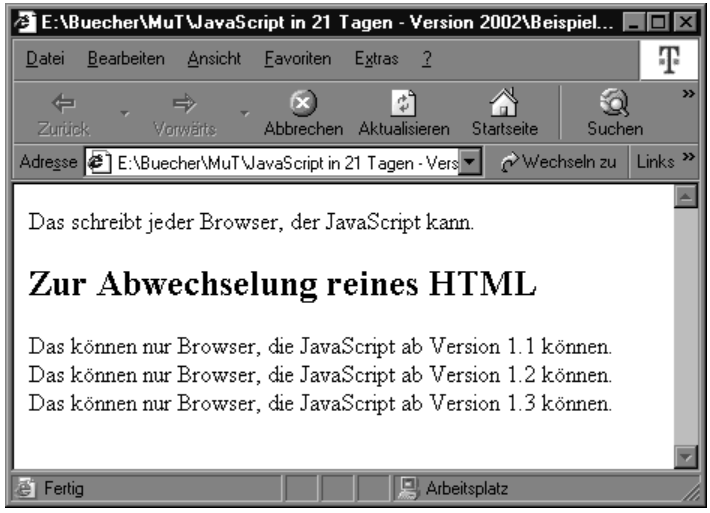

Abbildung 3.4: Ich kenne keine Version JavaScript1.0. Und was ist JavaScript 1.4 oder 1.5?

Das letzte Beispiel können Sie verwenden, um zu testen, welche JavaScript-Version ein Browser versteht, wenn Sie das Script in einen Browser laden. Sie sehen beispielsweise, dass der Internet Explorer 6 (nach seiner eigenen Einschätzung) nur JavaScript 1.3 beherrscht, während Opera 6 und Mozilla sich die Version 1.4 zutrauen und der Netscape Navigator 7 gar 1.5. Das Verfahren ist jedoch nur als eine untere Abschätzung zu gebrauchen, denn die meisten Browser können mehr, als sie sich hier zutrauen. Siehe dazu aber auch Anhang A und Tag 6, Abschnitt 6.7.

3.3 Externe Notation von Scripten

Neben der Notation von Scripten in den HTML-Quelltext selbst lassen sich JavaScripte unter HTML 4.0 und seit JavaScript in der Version 1.1 in einer oder mehreren externen Datei(en) ablegen. In dem Fall enthält nicht die Webseite den JavaScript-Code, sondern die externe Datei(en). In der Webseite wird nur der Aufruf einer Funktion durchgeführt. Um die Verbindung zwischen der Webseite und einer externen Datei herzustellen, muss in der Webseite der Pfad zu dieser Datei angegeben werden. Dies erfolgt mit einer Referenz, welche dann die Möglichkeit zum Laden des JavaScript-Codes eröffnet, wenn eine Funktion aufgerufen wird.

Die Referenz sieht für Scripte allgemein wie folgt aus:

```
<script language="[Sprache]" src="[URL der separaten Script-Datei]"> </script>
```

Im Fall von JavaScript gilt also:

```
<script language="JavaScript" src="[URL der separaten JavaScript-Datei]"></script>
```

Mit dem Attribut `src` wird der URL der separaten Datei angegeben. Dabei gelten beim Referenzieren separater JavaScript-Dateien die üblichen Regeln für URLs. Die separate JavaScript-Datei mit dem Quellcode muss wie HTML-Dateien auch eine reine Textdatei sein und ausschließlich JavaScript-Code enthalten. Es gibt kein `<script>`-Grundgerüst wie bei einer HTML-Seite. Üblich ist die Dateierweiterung *.js*, die aber nicht zwingend ist. Optional kann wieder das Attribut `type="text/JavaScript"` angegeben werden.

Der `<script>`-Container muss für den Fall einer Einbindung von externen Dateien **leer** sein! Diverse Browser reagieren sonst mit Fehlern. Alle JavaScript-Anweisungen befinden sich in der referenzierten Datei oder in mehreren referenzierten externen Dateien!

Externe Script-Dateien bedeuten zwar etwas mehr Verwaltungsaufwand, sind aber trotzdem oft sinnvoll. Folgende Argumente sprechen beispielsweise dafür:

- Gleiche JavaScript-Funktionalität soll in verschiedenen HTML-Dateien verwendet werden. Statt gleichen Code mehrfach in verschiedene Dateien zu notieren, muss nur jeweils eine Referenz auf eine zentrale Datei angegeben werden.
- Nur eine zentrale Stelle ist bei Wartungen des Codes erheblich sinnvoller. Wenn die Funktionalität mehrfach verwendet wird, muss dort keine Veränderung vorgenommen werden.
- Die Orientierung ist in einer reinen JavaScript-Datei einfacher.
- Die zu übertragende Datenmenge wird gegenüber dem Fall, in dem sich gleicher Code in mehreren HTML-Dateien befindet, reduziert.
- Analysen der Scripte durch einfaches Ansehen des Quellcodes der Webseite sind nicht mehr so einfach möglich (die Funktionalität befindet sich ja in einer anderen Datei, die man erst zusätzlich laden muss). Nicht immer ist gewünscht, dass man selbst lange an Scripten herumfeilt und dann jeder den Quellcode für sich verwendet.
- Eine logische Trennung von Dokumentenbestandteilen ist gewährleistet.
- Der Container wird von JavaScript-inkompatiblen Browsern ignoriert und da im Inneren keine Anweisungen stehen, kann es nicht zu Problemen kommen. Man vermeidet die Kommentarkonstrukte.
- Man kann Probleme in XHTML mit einigen Sonderzeichen umgehen.

Die Verwendung externer JavaScript-Dateien kann ein Problem bedeuten, wenn auf dem Server der MIME-Typ für Dateien mit der Endung der externen Datei (etwa *.js*) in der Konfiguration des Webservers nicht angegeben ist. Dann muss der JavaScript-Text direkt in eine HTML-Datei geschrieben werden. Auf jeden Fall sollte die JavaScript-Funktionalität nach dem Laden auf den Server getestet werden.

Beispiel für die externe Einbindung von JavaScript

Führen wir nun ein kleines Beispiel durch, das eine externe Script-Datei verwendet. Beachten Sie, dass hier zwei Funktionen verwendet werden, die beide in der Script-Datei abgelegt werden. Erstellen Sie zuerst eine Datei, in der die JavaScript-Funktionen definiert sind. Geben Sie im Editor den nachfolgenden Quelltext ein:

Listing 3.3: Die externe Textdatei, welche zwei JavaScript-Funktionen enthält

```
01 function willkommen() {
02 alert("Herzlich Willkommen");
03 }
04 function bey() {
05 alert("Und tschüss");
06 }
```

Speichern Sie den Text bitte unter dem Dateinamen *Listing3.js*. Die Zeilen 1 bis 3 sind die erste Funktion und die zweite Funktion erstreckt sich von Zeile 4 bis 6. Die jeweils aufgerufene Anweisung `alert()` öffnet ein kleines Mitteilungsfenster, dass die in Klammern angegebene Meldung anzeigt.

Das `alert`-Fenster wird sich je nach verwendetem Browser bezüglich der Optik unterscheiden.

Erstellen Sie nun die Datei, in der die Datei *Listing3.js* verwendet wird.

Listing 3.4: Die HTML-Datei mit der Referenz auf die externe Datei

```
01 <html>
02   <head>
03     <script language="JavaScript" src="Listing3.js"></script>
04   </head>
05   <body onLoad="willkommen()" onUnload="bey()">
06     Da schau an!
07   </body>
08 </html>
```

Beachten Sie, dass die Script-Referenz dieses Mal im Header notiert ist (Zeile 2 und 4). Die Zeile 3 ist die Referenz auf die externe Datei. In Zeile 5 wird mit zwei Eventhandlern[3] (`onLoad` und `onUnload`) gearbeitet. Diese rufen jeweils eine der beiden Funktionen aus der externen Datei auf.

Laden Sie die Datei in einen Browser, nachdem Sie sich zuvor eine andere Datei angesehen haben. Sie bekommen einen Begrüßungsdialog angezeigt.
Das macht `onLoad="Willkommen()"`.

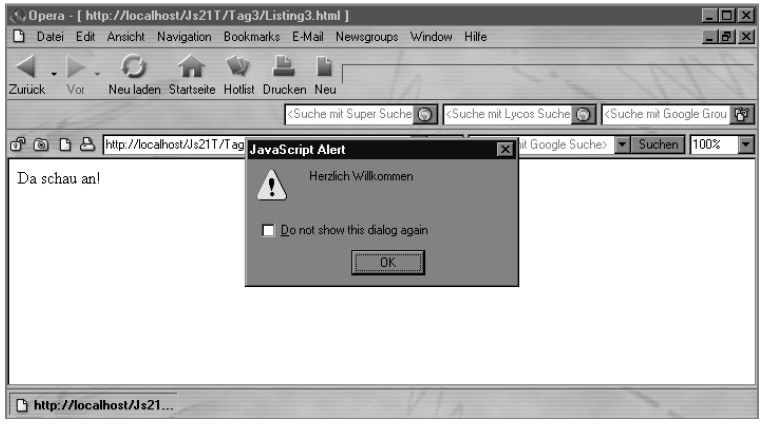

Abbildung 3.5: Laden der Webseite – hier im Opera 6

Verlassen Sie dann den Browser über die ZURÜCK-Schaltfläche. Sowohl beim Laden als auch beim Verlassen der Webseite wird eine JavaScript-Funktion aufgerufen, die jeweils eine unterschiedliche Meldung auf dem Bildschirm anzeigt.

Die Verwendung eines externen JavaScripts darf nicht so verstanden werden, dass alle dort definierten Funktionen auch verwendet werden müssen. Eine externe Datei kann beliebig viele Funktionen enthalten (und auch Variablen, welche wir noch nicht einsetzen), von denen bei Bedarf die angegebene Funktion geladen und verwendet wird. Die nicht benötigten Funktionen beeinträchtigen die Webseite in keiner Weise. Es sollte aber beachtet werden, dass eine externe JavaSript-Datei nicht unnötig groß wird. Eine Aufteilung auf mehrere externe Dateien ist u.U. sinnvoll und wird einfach dadurch möglich, dass die einzelnen, leeren Container mit den verschiedenen Referenzen gemeinsam in der Seite notiert werden. Beachten sollten Sie auch, dass JavaScript-Anweisungen, die direkt in eine externe Datei geschrieben werden (also ohne dass sie in einer Funktion eingebettet sind), beim Laden der Webseite mit der Referenz auf die externe JavaScript-Datei automatisch ausgeführt werden. Genau so, als ob sie direkt in der HTML-Datei stehen würden.

3 Siehe dazu Kapitel 11.

3.4 Der Inline-Aufruf von Script-Anweisungen

Script-Aufrufe lassen sich auch direkt in ein Attribut in einem HTML-Tag hineinschreiben. Das nennt man einen Inline-Aufruf oder auch Inline-Referenz, die in der Regel in Verbindung mit einer Steueranweisung verwendet wird, die auf einen Klick reagieren kann (etwa ein Hyperlink). Letzteres muss so vorsichtig beschrieben werden, weil einige Browser (insbesondere Netscape-Browser) diese Variante nur sehr eingeschränkt unterstützen und – natürlich – Browser ohne Script-Unterstützung mit der Technik nicht zurechtkommen. Der Internet Explorer kann diese Technik recht zuverlässig nutzen. Grundsätzlich sieht der Inline-Aufruf eines Scripts syntaktisch so aus:

```
<[verweissensitives Steuerelement] = "[Script-Sprache]:[Script-Anweisung]"> ...
<[Ende-Tag]>
```

Eine Webseite kommt sogar ganz ohne den `<script>`-Container aus, wenn bei einem Inline-Aufruf nur wenige vorgefertigte JavaScript-Anweisungen aufgerufen werden. Führen wir ein Beispiel mit einem Inline-Aufruf durch.

Ein Beispiel

Listing 3.5: Eine Inline-Referenz

```
01 <html>
02   <script language="JavaScript">
03     function zufall() {
04       alert(Math.random());
05     }
06   </script>
07   <body>
08     <a href="javascript:zufall()">Bitte, bitte klick mich</a>
09   </body>
10 </html>
```

Speichern Sie den Text und schauen Sie sich die Datei im Browser an. Klicken Sie mehrfach auf den Hyperlink. Das Beispiel verwendet einen Zufallsgenerator (in Zeile 4 – `Math.random()`), der ständig neue Zufallszahlen zwischen 0 und 1 generiert. Diese werden dann in einem Dialogfenster angezeigt. In Zeile 8 sehen Sie den Hyperlink, der mit einer Inline-Referenz die Funktion `zufall()` aufruft.

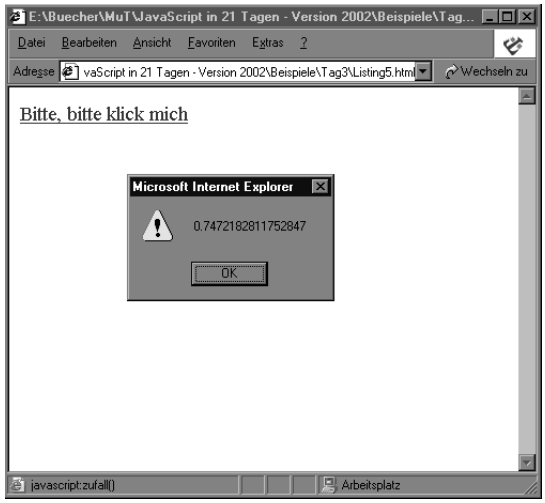

Abbildung 3.6:
Ein Klick auf den Hyperlink gibt eine zufällig generierte Zahl zwischen 0 und 1 aus.

Abbildung 3.7:
Ein neuer Klick auf den Link erzeugt eine andere Zufallszahl.

3.5 Spezielle Microsoft-Notation von Scripten für Events in einer HTML-Steueranweisung

Microsoft stellt für den Internet Explorer ein Eventmodell bereit, über das auf Ereignisse auf nahezu jedem Element in einer Webseite reagiert werden kann. Dazu wird der `<script>`-Tag dann zusammen mit einem `for`- und einem `event`-Attribut eingesetzt. Die Syntax sieht dann so aus:

```
<script for="[HTML-Steuerelement]" event="[Eventhandler]"
language="[Script-Sprache]">
 ... irgendwelche Script-Anweisungen
</script>
```

Diese Syntax wird von den meisten anderen Browsern nicht verstanden und bietet sich deshalb hauptsächlich für die Webseiten an, welche nur im Internet Explorer angezeigt werden sollen. Wir werden in einem späteren Teil des Buchs (siehe Kapitel 11.5.3 und Listing 11.10) eine praktische Anwendung zeigen.

3.6 Zusammenfassung

Sie kennen nun die Möglichkeiten zum Einbinden von JavaScripten (genau genommen von jeglichen clientseitigen Scripten, die im Rahmen eines Webbrowsers unterstützt werden) in Webseiten. Scripte werden mit HTML bzw. einer Webseite verknüpft. Entweder werden:

- die Quelltexte direkt in einen Container in einer Webseite notiert,
- eine externe Datei mit der Webseite verbunden oder
- eine Inline-Referenz verwendet.

Die konkrete Syntax für einen JavaScript-Container sieht also schematisch immer so aus:

```
<script language="JavaScript">
<!--
      ... irgendwelche Script-Anweisungen
//-->
</script>
```

Bei der externen Notation sieht die Referenz für Scripte allgemein wie folgt aus:

```
<script language="[Sprache]" src="[URL der separaten Script-Datei]"></script>
```

Im Fall von JavaScript gilt also Folgendes:

```
<script language="JavaScript" src="[URL der separaten JavaScript-Datei]"></script>
```

Mit dem Attribut `src` wird der URL der separaten Datei angegeben. Der `<script>`-Container muss für den Fall einer Einbindung von externen Dateien leer sein.

Der Inline-Aufruf eines Scripts sieht syntaktisch grundsätzlich so aus:

```
<[verweissensitives Steuerelement] = "[Script-Sprache]:[Script-Anweisung]"> ...
<[Ende-Tag]>
```

Für JavaScript also so:

```
<[verweissensitives Steuerelement] = "JavaScript:[Script-Anweisung]"> ...
<[Ende-Tag]>
```

3.7 Workshop

Fragen und Antworten

F *Wenn man ein Projekt beginnt, in dem Style Sheets und JavaScript verwendet werden, wird ja deutlich im Buch empfohlen, ab einer gewissen Komplexität externe Dateien zu verwenden. Ist es sinnvoll, jeden Dateityp in einem eigenen Verzeichnis abzulegen?*

A Ja, aber natürlich nicht zwingend. Dann sollte man aber auch so konsequent sein, für Grafiken und andere verwendete Ergänzungstechniken wie Java-Applets eigene Verzeichnisse zu führen. Ab einer gewissen Komplexität eines Webprojekts zahlt sich eine solche Organisation in Verzeichnissen aus.

F *Ab wann lohnt die Verwendung externer JavaScript-Dateien?*

A Eigentlich immer dann, wenn man eine Funktionalität mehr als einmal verwenden oder sein Projekt klar strukturieren will, und auch beim Einsatz von XHTML als Basis. Externe Script-Dateien sind eigentlich fast immer sinnvoll.

F *Gibt es noch weitere Möglichkeiten zur Script-Einbindung?*

A Ja, aber kaum einen Grund, sich damit zu beschäftigen.

F *Gibt es funktionale Unterschiede, wie ein extern eingebundenes Script gegenüber einem direkt in eine Webseite notierten Script arbeitet?*

A Nein.

F *Gibt es bereits vernünftige Anwendungen von Erweiterungen, die mit JavaScript 1.5 eingeführt wurden und die in einen entsprechend gekennzeichneten Container eingebunden werden müssen?*

A Ja. Da ist einmal das so genannte Ausnahmebehandlungskonzept, das zum qualifizierten Reagieren auf bestimmte Fehlersituationen genutzt werden kann (das werden wir an Tag 7 behandeln). Aber auch die Verwendung von dem in JavaScript 1.5 eingeführten Schlüsselwort `const` eröffnet sinnvolle Möglichkeiten, der ungewollten Veränderung einer Variablen entgegenzusteuern. Das werden wir morgen zeigen.

Quiz

In Anhang B finden Sie die Antworten.

1. Finden Sie den Fehler:
   ```
   <script language ="JavaScript 1.2">
   ```
2. Finden Sie den Fehler:
   ```
   <script langauge="JavaScript" src="extern.js"> </script>
   ```
3. Finden Sie den Fehler:
   ```
   <A href="zufall()">Klick</A>
   ```
4. Finden Sie den Fehler:
   ```
   <script language="JavaScript" src="extern.js">alert("Hallo");</script>
   ```
5. Was stimmt an dieser MIME-Angabe nicht?
   ```
   text\javascript
   ```

Übungen

Experimentieren Sie mit einfachen HTML-Seiten, indem Sie unsere externen JavaScript-Dateien dort einbinden und verwenden.

Gehen Sie online und analysieren Sie Webseiten mit JavaScripten auf deren Weg der Einbindung. Den Quelltext einer Webseite können Sie sich im Navigator und im Internet Explorer über das ANSICHT-Menü und dort den jeweiligen Menüpunkt SEITENQUELLTEXT bzw. QUELLTEXT ANZEIGEN ansehen. Bei Frames – auf die gehen wir noch ein – sehen Sie unter Umständen nicht viel. Klicken Sie dann einfach mit der rechten Maustaste auf den für Sie interessanten Bereich und wählen Sie aus dem Kontextmenü den Befehl zum Anzeigen des Teilbereichs aus.