



# jQuery lernen und einsetzen

Bessere Webanwendungen mit einfachen  
JavaScript-Techniken entwickeln

*Learning jQuery*  
Deutsche Ausgabe  
der 3. engl. Aufl.

dpunkt.verlag

## **jQuery lernen und einsetzen**



**Jonathan Chaffer · Karl Swedberg**

# **jQuery lernen und einsetzen**

**Bessere Webanwendungen mit einfachen  
JavaScript-Techniken entwickeln**

Übersetzung der 3. engl. Auflage



**dpunkt.verlag**

Lektorat: Dr. Michael Barabas

Copy Editing: Ursula Zimpfer, Herrenberg

Übersetzung und Satz: G&U Language & Publishing Services GmbH, Flensburg, ([www.GundU.com](http://www.GundU.com))

Herstellung: Nadine Thiele

Umschlaggestaltung: Helmut Kraus, [www.exclam.de](http://www.exclam.de)

Druck und Bindung: M.P. Media-Print Informationstechnologie GmbH, 33100 Paderborn

#### 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.

ISBN:

Buch 978-3-89864-786-1

PDF 978-3-86491-150-7

ePub 978-3-86491-151-4

1. Auflage 2012

Copyright der deutschen Ausgabe © 2012 [dpunkt.verlag](http://dpunkt.verlag) GmbH

Ringstraße 19B

69115 Heidelberg

Copyright © Packt Publishing 2011.

First published in the English language under the title »Learning jQuery«, Third Edition

Die vorliegende Publikation ist urheberrechtlich geschützt. Alle Rechte vorbehalten. Die Verwendung der Texte und Abbildungen, auch auszugsweise, ist ohne die schriftliche Zustimmung des Verlags urheberrechtswidrig und daher strafbar. Dies gilt insbesondere für die Vervielfältigung, Übersetzung oder die Verwendung in elektronischen Systemen.

Es wird darauf hingewiesen, dass die im Buch verwendeten Soft- und Hardware-Bezeichnungen sowie Markennamen und Produktbezeichnungen der jeweiligen Firmen im Allgemeinen warenzeichen-, marken- oder patentrechtlichem Schutz unterliegen.

Alle Angaben und Programme in diesem Buch wurden mit größter Sorgfalt kontrolliert. Weder Autor noch Verlag können jedoch für Schäden haftbar gemacht werden, die in Zusammenhang mit der Verwendung dieses Buches stehen.

5 4 3 2 1 0

---

## Geleitwort

Ich fühle mich geehrt, dass Karl Swedberg und Jonathan Chaffer die Mühe auf sich genommen haben, »Learning jQuery« zu schreiben. Als erstes jQuery-Buch setzte es den Standard, den andere jQuery-Bücher – und praktisch auch alle anderen JavaScript-Bücher – zu erreichen versuchen. Es ist eines der meistgekauften JavaScript-Bücher, was zu keinem geringen Teil seiner Genauigkeit und seinen vielen Details zu verdanken ist.

Er freut mich besonders, dass Karl und Jonathan dieses Buch geschrieben haben, weil ich sie bereits so gut kannte, dass ich wusste, sie wären die besten für diese Aufgabe. Als Teil des Kernteam von jQuery hatte ich Karl über mehrere Jahre intensiv kennen gelernt und natürlich besonders, während dieses Buch entstand. Wenn ich das Ergebnis betrachte, wird deutlich, dass seine Fähigkeiten als Entwickler und als ehemaliger Englischlehrer hier eine hervorragende Kombination abgegeben haben.

Beide Autoren konnte ich persönlich treffen, was in der Welt verteilter Open-Source-Projekte eher selten ist, und beide sind und bleiben herausragende Mitglieder der jQuery-Gemeinschaft.

Die jQuery-Bibliothek wird von den unterschiedlichsten Personen der jQuery-Community genutzt. Diese Gemeinschaft ist voll von Designern, Entwicklern, Personen mit Programmiererfahrung oder ohne. Auch im jQuery-Team finden sich Personen mit den unterschiedlichsten beruflichen Hintergründen, die alle ihren Beitrag zum Projekt leisten. Eine Sache ist jedoch allen jQuery-Anwendern gemein: Wir sind eine Gemeinschaft von Entwicklern und Designern, die sich wünscht, dass die JavaScript-Entwicklung einfach wird.

Es ist an dieser Stelle fast schon klischeehaft zu sagen, dass ein Open-Source-Projekt gemeinschaftsorientiert ist oder dass es sich darauf fokussiert, neuen Anwendern den Einstieg zu erleichtern. Bei jQuery ist das jedoch keine leere Floskel, sondern die treibende Kraft des Projekts. Es gibt momentan mehr Personen im jQuery-Team, die sich der Gemeinschaft, der Dokumentation und den Plugins widmen, als der Wartung der Codebasis. Obwohl eine korrekte Bibliothek

extrem wichtig ist, macht die Gemeinschaft den Unterschied zwischen einem stöckenden, mittelmäßigen und einem Projekt aus, das allen Anforderungen gerecht wird oder sie noch übertrifft.

Wie wir das Projekt betreiben und wie Sie den Code einsetzen, unterscheidet sich deutlich von anderen Open-Source-Projekten und den meisten JavaScript-Bibliotheken. Das jQuery-Projekt und die Community verfügen über weitreichende Sachkunde, wir wissen, warum jQuery eine völlig andere Art zu Programmieren ist, und tun unser bestes, dieses Wissen weiterzugeben.

Die jQuery-Gemeinschaft können Sie nicht durch bloßes Lesen verstehen, Sie müssen daran teilhaben, um vollständig darin einzutauchen. Ich hoffe, Sie haben die Möglichkeit dazu. Nehmen Sie an den Foren, Mailinglisten und Blogs teil, und lassen Sie uns Ihnen dabei helfen, jQuery besser kennen zu lernen.

Für mich ist jQuery viel mehr als ein Haufen Quelltext. Es ist die Summe der über die Jahre angesammelten Erfahrungen, die eine Bibliothek bilden, die beträchtlichen Höhen und Tiefen, die schwierige Entwicklung und die aufregenden Momente, das Projekt wachsen und blühen zu sehen. Und es ist ein Zusammenwachsen der Anwender und der Team-Mitglieder, das Bemühen, sie zu verstehen, und der Versuch, sich anzupassen und zu wachsen.

Als ich dieses Buch erstmals gesehen und seinen Ansatz, jQuery als einheitliches Werkzeug zu erläutern, gelesen habe, war ich verblüfft und angeregt zugleich. Zu erkennen wie andere lernen, verstehen und jQuery so anpassen, wie es für sie richtig ist, macht das Besondere dieses Projekts aus.

Ich bin nicht der einzige, der jQuery von einer anderen Warte als der normalen Werkzeug-Anwender-Beziehung betrachtet. Ich bin nicht sicher, ob ich genau sagen kann, warum, aber ich habe es wieder und wieder beobachtet – der Augenblick, in dem der Anwender zu lächeln beginnt, weil er erkennt, wie viel jQuery ihm helfen kann.

Es gibt diesen besonderen Moment, bei dem es beim Anwender »Klick macht«, wenn er erkennt, dass dieses Werkzeug viel, viel mehr ist als ein einfaches Hilfsmittel. In diesem Augenblick ändert sich auch sein Verständnis völlig, wie dynamische Webanwendungen zu programmieren sind. Das ist eine faszinierende Sache und immer der schönste Augenblick in einem jQuery-Projekt.

Ich hoffe, dass Sie dieses Erlebnis ebenfalls haben werden.

John Resig  
*Erfinder von jQuery*

## Über die Autoren

**Jonathan Chaffer** ist Mitglied der Rapid Development Group, einer Web-Entwicklungsfirma in Grand Rapids, Michigan. Zu seiner Arbeit gehört die Projektüberwachung und Implementierung einer Vielzahl von Technologien mit einem Schwerpunkt auf PHP, MySQL und JavaScript. Er leitet außerdem jQuery-Trainingsseminare für Webentwickler.

In der Open-Source-Gemeinschaft ist Jonathan sehr aktiv beim Drupal-CMS-Projekt, das jQuery als JavaScript-Framework ausgewählt hat. Er ist der Schöpfer des Content Construction Kits, eines beliebten Moduls für die Verwaltung strukturierter Inhalte auf Drupal-Sites. Er ist für die wesentlichen Überarbeitungen im Drupal-Menüsystem und der Entwickler-API verantwortlich.

Jonathan lebt mit seiner Frau Jennifer in Grand Rapids.

Ich möchte Jenny für ihre unermüdliche Begeisterung und Unterstützung danken, Karl für die Motivation weiterzumachen, als es beim Skript langsamer voranging, und der Ars-Technica-Gemeinschaft für ihre fortwährende Inspiration bei diesem technischen Werk. Zusätzlich möchte ich Mike Henry und dem Twisted Pixel-Team für die netten Ablenkungen zwischen meinen Schreibphasen danken.

**Karl Swedberg** ist Webentwickler bei Fusionary Media in Grand Rapids, Michigan, wo er viel Zeit damit verbringt, tolle Sachen mit JavaScript anzustellen. Als Mitglied des jQuery-Teams ist Karl zuständig für die Pflege der jQuery-API-Site unter [api.jquery.com](http://api.jquery.com). Er veröffentlicht außerdem Anleitungen in seinem Blog [learningjquery.com](http://learningjquery.com) und präsentiert auf Workshops und Konferenzen. Wenn er nicht gerade programmiert, verbringt Karl gern seine Zeit mit der Familie, röstet in seiner Garage Kaffee und trainiert im örtlichen Fitnessclub.

Ich danke meiner Frau Sara und meinen beiden Kindern Benjamin und Lucia für all die Freude, die sie meinem Leben geben. Dank auch an Jonathan Chaffer für seine Geduld und die gemeinsame Arbeit an diesem Buch.

Vielen Dank auch an John Resig für die beste JavaScript-Bibliothek der Welt und alle anderen, die ihren Code, ihre Zeit und Expertise in dieses Projekt investiert haben. Danke an die Mitarbeiter von Packt Publishing, die technischen Redakteure dieses Buchs, jQuery Cabal und den vielen anderen, die auf dem langen Weg Hilfe und Inspiration geleistet haben.

## Die Fachgutachter

**Kaiser Ahmed** ist professioneller Webentwickler mit einem Bachelor-Grad von der Khula University of Engineering and Technology (KUET). Außerdem ist er Mitbegründer des vollständig auf Outsourcing gestützten Unternehmens CyberXpress.Net Inc. in Bangladesch.

Er verfügt über ein breites Spektrum an technischen Fertigkeiten, Kenntnissen über das Internet und Erfahrung über die ganze Palette der Onlineentwicklung hinweg, die er dazu einsetzt, Onlinepräsentationen für viele Kunden zu erstellen und zu verbessern. Er hat viel Freude an der Gestaltung der Architektur und Infrastruktur von Websites, an der Back-End-Entwicklung mit Open-Source-Werkzeugen (Linux, Apache, MySQL, PHP [LAMP] und andere) und an der Front-End-Entwicklung mit CSS und HTML/XHTML.

*Er möchte seiner liebenden Frau Maria Akter  
für ihre Unterstützung danken.*

**Kevin Boudloche** ist ein Webentwickler aus Mississippi. Webseiten erstellt er als Hobby seit mehr als acht Jahren und professionell seit drei Jahren. Sein Schwerpunkt liegt auf der Entwicklung von Front-Ends und Webanwendungen.

**Carlos Esteves** ist der Gründer von Ehxioz (<http://ehxioz.com/>), einer jungen Softwareentwicklungs firma aus Los Angeles, die sich auf die Entwicklung moderner Webanwendungen spezialisiert hat und die neuesten Technologien und Methoden zur Webentwicklung nutzt. Er hat über zehn Jahre Erfahrung als Webentwickler und einen Bachelor-Grad in Informatik von der California State University in Los Angeles.



# Inhaltsverzeichnis

<b>Einleitung</b>	<b>1</b>
<b>1 Erste Schritte</b>	<b>7</b>
1.1 Was bietet jQuery? .....	7
1.2 Warum jQuery so gut funktioniert .....	9
1.3 Unsere erste Webseite mit jQuery .....	11
1.3.1 jQuery herunterladen .....	11
1.3.2 Einrichten von jQuery in einem HTML-Dokument .....	11
1.3.3 jQuery-Code hinzufügen .....	14
1.3.4 Das fertige Produkt .....	16
1.4 Einfaches JavaScript und jQuery im Vergleich .....	17
1.5 Entwicklungswerzeuge .....	18
1.5.1 Firebug .....	19
1.6 Zusammenfassung .....	22
<b>2 Elemente auswählen</b>	<b>23</b>
2.1 Das Document Object Model .....	23
2.2 Die Funktion \$() .....	25
2.3 CSS-Selektoren .....	26
2.3.1 Listenelemente formatieren .....	27
2.3.2 Attributselektoren .....	29
2.3.3 Links formatieren .....	29
2.4 jQuery-Selektoren .....	31
2.4.1 Zeilen abwechselnd formatieren .....	32
2.4.2 Formularselektoren .....	36
2.5 Methoden zum Durchlaufen des DOM .....	37
2.5.1 Einzelne Zellen formatieren .....	38
2.5.2 Verkettung .....	40

2.6	Zugriff auf DOM-Elemente .....	41
2.7	Zusammenfassung .....	42
2.7.1	Literatur .....	42
2.8	Übungsaufgaben .....	43
<b>3</b>	<b>Ereignisbehandlung</b>	<b>45</b>
3.1	Aufgaben beim Laden der Seite durchführen .....	45
3.1.1	Die Codeausführung zeitlich abstimmen .....	45
3.1.2	Mehrere Skripte auf einer Seite .....	46
3.1.3	Kurzschreibweisen .....	48
3.1.4	Argumente an den .ready()-Callback übergeben .....	48
3.2	Einfache Ereignisse .....	49
3.2.1	Ein einfacher Formatwechsler .....	49
3.2.2	Die anderen Schaltflächen aktivieren .....	52
3.2.3	Ereignishandler-Kontext .....	53
3.2.4	Weitere Konsolidierung .....	55
3.2.5	Kurzformen für Ereignisse .....	57
3.3	Zusammengesetzte Ereignisse .....	58
3.3.1	Erweiterte Funktionen anzeigen und ausblenden .....	58
3.3.2	Anklickbare Elemente hervorheben .....	60
3.4	Der Weg eines Ereignisses .....	62
3.4.1	Nebenwirkungen des Event Bubbling .....	64
3.5	Den Weg ändern: das Ereignisobjekt .....	64
3.5.1	Ereignisziele .....	66
3.5.2	Die Ereignisweiterleitung abbrechen .....	66
3.5.3	Standardaktionen .....	67
3.5.4	Ereignisdelegierung .....	68
3.5.5	Methoden für die Ereignisdelegierung .....	71
3.6	Ereignishandler entfernen .....	71
3.6.1	Namensräume für Ereignisse .....	72
3.6.2	Ereignisse erneut binden .....	73
3.7	Benutzerinteraktion simulieren .....	75
3.7.1	Tastaturereignisse .....	76
3.8	Zusammenfassung .....	79
3.8.1	Literatur .....	80
3.9	Übungsaufgaben .....	80

<b>4</b>	<b>Formatierung und Animation</b>	<b>81</b>
4.1	Inline-Bearbeitung mit CSS .....	81
4.2	Anzeigen und Verbergen .....	86
4.3	Effekte und Speed .....	89
4.3.1	Anzeigen mit »Geschwindigkeit« .....	89
4.3.2	Ein- und ausblenden .....	90
4.3.3	Auseinander- und zusammenfalten .....	91
4.3.4	Zusammengesetzte Effekte .....	92
4.4	Benutzerdefinierte Animationen erstellen .....	93
4.4.1	Effekte manuell erstellen .....	94
4.4.2	Mehrere Eigenschaften gleichzeitig animieren .....	95
4.5	Gleichzeitige und aneinandergereihte Effekte .....	99
4.5.1	Mit einem einzelnen Satz von Elementen arbeiten .....	99
4.5.2	Mit mehreren Sätzen von Elementen arbeiten .....	103
4.5.3	Kurz und bündig .....	107
4.6	Zusammenfassung .....	108
4.6.1	Literatur .....	108
4.7	Übungsaufgaben .....	108
<b>5</b>	<b>DOM-Bearbeitung</b>	<b>109</b>
5.1	Attribute bearbeiten .....	109
5.1.1	Nicht-Klassenattribute .....	110
5.1.2	Eigenschaften von DOM-Elementen .....	113
5.2	Bearbeitung des DOM-Baums .....	114
5.2.1	Neues zur Funktion \$() .....	114
5.2.2	Neue Elemente erstellen .....	115
5.2.3	Neue Elemente einfügen .....	116
5.2.4	Elemente verschieben .....	117
5.2.5	Elemente verschachteln .....	119
5.2.6	Umgekehrte Einfügemethoden .....	121
5.3	Elemente kopieren .....	125
5.3.1	Klonen für interne Zitate .....	126
5.4	Get- und Set-Methoden für Inhalte .....	128
5.4.1	Weitere Formatanpassungen .....	130
5.5	Methoden zur DOM-Bearbeitung – kurz und bündig .....	132
5.6	Zusammenfassung .....	133
5.6.1	Literatur .....	133
5.7	Übungsaufgaben .....	133

<b>6</b>	<b>Daten mit Ajax senden</b>	<b>135</b>
6.1	Daten bei Bedarf laden .....	135
6.1.1	HTML anhängen .....	137
6.1.2	Mit JavaScript-Objekten arbeiten .....	140
6.1.3	XML-Dokumente laden .....	146
6.2	Ein Datenformat auswählen .....	149
6.3	Daten an den Server übergeben .....	150
6.3.1	GET-Requests durchführen .....	151
6.3.2	POST-Requests durchführen .....	155
6.3.3	Formulare serialisieren .....	156
6.4	Unterschiedliche Inhalte liefern .....	158
6.5	Die Anforderung im Auge behalten .....	160
6.6	Fehlerbehandlung .....	162
6.7	Ereignisse in Ajax .....	164
6.8	Sicherheitseinschränkungen .....	165
6.8.1	JSONP für fremde Daten verwenden .....	166
6.9	Zusätzliche Optionen .....	168
6.9.1	Die grundlegende Methode ajax .....	168
6.9.2	Standardoptionen ändern .....	169
6.9.3	Teile einer HTML-Seite laden .....	170
6.10	Zusammenfassung .....	172
6.10.1	Literatur .....	172
6.11	Übungsaufgaben .....	173
<b>7</b>	<b>Plug-ins verwenden</b>	<b>175</b>
7.1	Plug-ins finden und Unterstützung bekommen .....	175
7.2	Ein Plug-in verwenden .....	176
7.2.1	Das Cycle-Plug-in herunterladen und einbinden .....	176
7.2.2	Einfache Plug-in-Anwendungen .....	176
7.2.3	Parameter an Plug-in-Methoden übergeben .....	178
7.2.4	Voreingestellte Parameter .....	179
7.2.5	Andere Arten von Plug-ins .....	180
7.3	Die UI-Plug-in-Bibliothek von jQuery .....	182
7.3.1	Effekte .....	182
7.3.2	Interaktionskomponenten .....	186
7.3.3	Widgets .....	187
7.3.4	JQuery-UI-ThemeRoller .....	190
7.4	Zusammenfassung .....	191
7.5	Übungsaufgaben .....	191

<b>8</b>	<b>Plug-ins entwickeln</b>	<b>193</b>
8.1	Das Alias \$ innerhalb von Plug-ins verwenden .....	193
8.2	Neue globale Funktionen hinzufügen .....	194
8.2.1	Mehrere Funktionen hinzufügen .....	196
8.3	JQuery Objektmethoden hinzufügen .....	199
8.3.1	Kontext von Objektmethoden .....	200
8.3.2	Implizite Iteration .....	201
8.3.3	Verkettete Methoden .....	202
8.4	Methodenparameter .....	203
8.4.1	Parameter-Maps .....	204
8.4.2	Voreinstellungen für Parameterwerte .....	205
8.4.3	Callback-Funktionen .....	206
8.4.4	Anpassbare Voreinstellungen .....	208
8.5	Die Widget-Factory von jQuery UI .....	209
8.5.1	Ein Widget erstellen .....	210
8.5.2	Widgets entfernen .....	212
8.5.3	Widgets aktivieren und deaktivieren .....	213
8.5.4	Widget-Optionen übernehmen .....	213
8.5.5	Untermethoden hinzufügen .....	214
8.5.6	Widget-Ereignisse auslösen .....	215
8.6	Designempfehlungen für Plug-ins .....	216
8.6.1	Plug-ins veröffentlichen .....	217
8.7	Zusammenfassung .....	217
8.8	Übungsaufgaben .....	218
<b>9</b>	<b>Komplexe Selektoren und Durchlaufen des DOM</b>	<b>219</b>
9.1	Auswahl und Durchlaufen – Teil 2 .....	219
9.1.1	Dynamisches Filtern von Tabellen .....	221
9.1.2	Streifenmuster für Tabellenzeilen .....	223
9.1.3	Filter und Streifenmuster kombinieren .....	225
9.1.4	Weitere Selektoren und Traversierungsmethoden .....	226
9.2	Selektoren anpassen und optimieren .....	226
9.2.1	Ein eigenes Selektor-Plug-in schreiben .....	226
9.2.2	Selektor-Performance .....	228
9.3	Durchlaufen des DOM – Hinter den Kulissen .....	231
9.3.1	jQuery-Objekteigenschaften .....	232
9.3.2	Der DOM-Elementstack .....	234
9.3.3	Ein Plug-in für DOM-Traversierungsmethoden schreiben .....	235
9.3.4	Performance von DOM-Traversierungsmethoden .....	237
9.4	Zusammenfassung .....	239
9.4.1	Literatur .....	239
9.5	Übungsaufgaben .....	239

<b>10 Komplexe Ereignisse</b>	<b>241</b>
10.1 Ereignisse – Teil 2 .....	241
10.1.1 Zusätzliche Datenseiten laden .....	243
10.1.2 Daten beim Darüberfahren mit der Maus anzeigen .....	244
10.2 Ereignisdelegation .....	245
10.2.1 Die jQuery-Delegationsmethoden verwenden .....	246
10.2.2 Eine Delegationsmethode wählen .....	247
10.2.3 Frühe Delegation .....	248
10.2.4 Ein Kontextargument verwenden .....	249
10.3 Benutzerdefinierte Ereignisse .....	249
10.3.1 Unendliches Scrollen .....	251
10.3.2 Benutzerdefinierte Ereignisparameter .....	252
10.4 Ereignisse drosseln .....	253
10.4.1 Andere Arten der Drosselung .....	254
10.5 Spezielle Ereignisse .....	255
10.5.1 Weitere Informationen zu speziellen Ereignissen .....	257
10.6 Zusammenfassung .....	257
10.6.1 Literatur .....	257
10.7 Übungsaufgaben .....	258
<b>11 Anspruchsvolle Effekte</b>	<b>259</b>
11.1 Animation – Teil 2 .....	259
11.2 Animationen beobachten und unterbrechen .....	261
11.2.1 Den Animationsstatus bestimmen .....	262
11.2.2 Eine laufende Animation anhalten .....	263
11.3 Globale Effekteigenschaften .....	264
11.3.1 Globales Deaktivieren aller Effekte .....	264
11.3.2 Feineinstellung der Animationsübergänge .....	265
11.3.3 Die Effektdauer festlegen .....	265
11.4 Easing mit mehreren Eigenschaften .....	268
11.5 Verzögerte Objekte .....	268
11.5.1 Animations-Promises .....	270
11.6 Zusammenfassung .....	273
11.6.1 Literatur .....	273
11.7 Übungsaufgaben .....	273

---

<b>12</b>	<b>DOM-Manipulation für Fortgeschrittene</b>	<b>275</b>
12.1	Tabellenzeilen sortieren .....	275
12.1.1	Serverseitiges Sortieren .....	275
12.1.2	Sortierung mit Ajax .....	276
12.1.3	Sortierung mit JavaScript .....	277
12.2	Elemente verschieben und einfügen – Teil 2 .....	278
12.2.1	Links um bestehenden Text herum einfügen .....	278
12.2.2	Einfache JavaScript-Arrays sortieren .....	279
12.2.3	DOM-Elemente sortieren .....	280
12.3	Daten zusammen mit DOM-Elementen ablegen .....	282
12.3.1	Zusätzliche Vorberechnungen .....	283
12.3.2	Nicht-String-Daten speichern .....	284
12.3.3	Umkehren der Sortierrichtung .....	286
12.4	HTML5 mit eigenen Datenattributen einsetzen .....	288
12.5	Zeilen mit JSON sortieren und erzeugen .....	290
12.5.1	Das JSON-Objekt modifizieren .....	292
12.5.2	Inhalte bei Bedarf wiederherstellen .....	293
12.6	Attributmanipulation für Fortgeschrittene .....	295
12.6.1	Elementerstellung per Kurzschrift .....	295
12.6.2	DOM-Manipulation mit Hooks .....	296
12.7	Zusammenfassung .....	298
12.7.1	Literatur .....	299
12.8	Übungsaufgaben .....	299
<b>13</b>	<b>Ajax für Fortgeschrittene</b>	<b>301</b>
13.1	Fortschreitende Verbesserung mit Ajax .....	301
13.1.1	JSONP-Daten einsammeln .....	303
13.2	Ajax-Fehlerbehandlung .....	306
13.3	Das jqXHR-Objekt .....	308
13.3.1	Ajax-Promises .....	309
13.3.2	Antworten cachen .....	310
13.4	Ajax-Anfragen drosseln .....	312
13.5	Ajax-Funktionen erweitern .....	313
13.5.1	Konverter für Datentypen .....	313
13.5.2	Ajax-Prefilter .....	318
13.5.3	Alternative Transporte .....	318
13.6	Zusammenfassung .....	322
13.6.1	Literatur .....	322
13.7	Übungsaufgaben .....	322

<b>A</b>	<b>JavaScript-Closures</b>	<b>323</b>
A.1	Innere Funktionen .....	323
A.1.1	Gesprengte Ketten .....	324
A.1.2	Gültigkeitsbereiche von Variablen .....	326
A.2	Interaktion zwischen Closures .....	328
A.3	Closures in jQuery .....	329
A.3.1	Argumente für \$(document).ready() .....	329
A.3.2	Ereignishandler .....	330
A.3.3	Handler in Schleifen binden .....	331
A.3.4	Benannte und anonyme Funktionen .....	333
A.4	Gefahren durch Speicherlecks .....	334
A.4.1	Unerwünschte Verweisschleifen .....	335
A.4.2	Internet Explorer und sein Speicherleck-Problem .....	336
A.5	Zusammenfassung .....	338
<b>B</b>	<b>JavaScript mit QUnit testen</b>	<b>339</b>
B.1	QUnit herunterladen .....	339
B.2	Das Dokument einrichten .....	340
B.3	Tests organisieren .....	341
B.4	Tests hinzufügen und ausführen .....	342
B.4.1	Asynchrones Testen .....	344
B.5	Andere Testarten .....	345
B.6	Praktische Erwägungen .....	346
B.6.1	Literatur .....	347
B.7	Zusammenfassung .....	347
<b>C</b>	<b>Kurzreferenz</b>	<b>349</b>
C.1	Selektorausdrücke .....	349
C.2	Methoden zum Durchlaufen des DOM .....	352
C.3	Ereignismethoden .....	354
C.4	Effektmethoden .....	357
C.5	DOM-Manipulationsmethoden .....	359
C.6	Ajax-Methoden .....	362
C.7	Verzögerte Objekte .....	364
C.8	Verschiedene Eigenschaften und Funktionen .....	365
	<b>Index</b>	<b>367</b>

---

# Einleitung

Angeregt von Pionieren auf diesem Gebiet wie Dean Edwards und Simon Willison, stellte John Resig 2005 einen Satz von Funktionen zusammen, um den Vorgang zu vereinfachen, durch Programme Elemente auf einer Webseite zu finden und ihnen Verhalten zuzuweisen. Als er sein Projekt im Januar 2006 erstmals der Öffentlichkeit vorstellte, hatte er ihm DOM-Bearbeitungsmöglichkeiten und einfache Animationen hinzugefügt. Er nannte es jQuery, um die zentrale Rolle hervorzuheben, die das Abfragen (*to query*) von Webseiten und ihre Bearbeitung mit JavaScript spielten. In den wenigen Jahren, die seither vergangen sind, wurde der Funktionsumfang von jQuery erweitert und die Leistung verbessert. Viele der beliebtesten Websites im Internet greifen inzwischen auf jQuery zurück. Resig bleibt zwar der leitende Entwickler, doch jQuery ist in echter Open-Source-Manier zu einem Projekt gewachsen, das sich eines Kernteam von JavaScript-Spitzenentwicklern und einer lebendigen Community von Tausenden von Entwicklern rühmen kann.

Die JavaScript-Bibliothek jQuery kann auch Ihre Websites unabhängig von Ihrem Hintergrund aufwerten. Sie bietet eine breite Palette von Funktionen, eine leicht zu erlernende Syntax und eine solide plattformübergreifende Kompatibilität in einer einzigen, kompakten Datei. Überdies wurden Hunderte von Plug-ins entwickelt, um den Funktionsumfang von jQuery zu erweitern und es zu einem unverzichtbaren Werkzeug für praktisch jede clientseitige Skriptaufgabe zu machen.

Dieses Buch gibt eine behutsame Einführung in die Prinzipien von jQuery, damit Sie Ihren Seiten Interaktion und Animationen hinzufügen können – auch wenn frühere Versuche, JavaScript zu schreiben, Sie nur in Verwirrung gestürzt haben. Dieses Buch hilft Ihnen, die Klippen zu umschiffen, die bei Ajax, Ereignissen, Effekten und anspruchsvollerem Merkmalen der Sprache JavaScript lauern. Außerdem fungiert es als kurzes Nachschlagewerk zur Bibliothek jQuery, die sie immer wieder benutzen können.

## Der Inhalt dieses Buches

In Kapitel 1, *Erste Schritte*, lernen Sie die JavaScript-Bibliothek jQuery kennen. Das Kapitel beginnt mit einer Beschreibung von jQuery und dem Nutzen für Sie. Anschließend erfahren Sie, wie Sie die Bibliothek herunterladen und einrichten und wie Sie Ihr erstes Skript schreiben.

In Kapitel 2, *Elemente auswählen*, lernen Sie, wie Sie die Selektorausdrücke und DOM-Durchquerungsmethoden von jQuery nutzen, um Elemente auf einer Seite zu finden, wo auch immer sie stecken mögen. Sie verwenden jQuery, um unterschiedliche Seitenelemente zu formatieren, teilweise sogar auf eine Weise, die mit reinem CSS nicht möglich ist.

In Kapitel 3, *Ereignisbehandlung*, nutzen Sie den Ereignisbehandlungsmechanismus von jQuery, um beim Auftreten bestimmter Browserereignisse Verhaltensweisen auszulösen. Sie erfahren, wie Sie mit jQuery auf unaufdringliche und einfache Weise Ereignisse an Elemente anhängen können, selbst wenn die Seite noch nicht vollständig geladen ist. Außerdem erhalten Sie einen Überblick über anspruchsvollere Themen wie Event Bubbling, Delegation und die Verwendung von Namensräumen.

In Kapitel 4, *Formatierung und Animation*, werden die Animationstechniken von jQuery eingeführt. Sie erfahren, wie Sie Seitenelemente mit sowohl nützlichen als auch ästhetischen Effekten ein- und ausblenden und verschieben können.

In Kapitel 5, *DOM-Bearbeitung*, lernen Sie, wie Sie Ihre Seite auf Befehl umgestalten können. Sie erfahren, wie Sie sowohl den Inhalt als auch die Struktur eines HTML-Dokuments im laufenden Betrieb ändern können.

In Kapitel 6, *Daten mit Ajax senden*, lernen Sie die verschiedenen Möglichkeiten kennen, mit denen jQuery den Zugriff auf serverseitige Funktionen ohne hinderliche Seitenaktualisierungen vereinfacht. Nachdem Sie nun die grundlegenden Bestandteile der Bibliothek beherrschen, können Sie sich genauer ansehen, wie Sie sie erweitern können, um sie an Ihre Bedürfnisse anzupassen.

Kapitel 7, *Plug-ins verwenden*, zeigt Ihnen, wie Sie Plug-ins finden, installieren und verwenden, u.a. die leistungsfähige Plug-in-Bibliothek jQuery UI.

In Kapitel 8, *Plug-ins entwickeln*, lernen Sie, wie Sie die eindrucksvollen Erweiterungsfähigkeiten von jQuery nutzen können, um eigene Plug-ins von Grund auf zu erstellen. Sie legen hier eigene Hilfsfunktionen an, fügen jQuery-Objektmethoden hinzu und sehen sich die Widget-Factory von jQuery UI näher an. Danach beschäftigen wir uns erneut mit den Grundbausteinen von jQuery, um einige anspruchsvollere Techniken zu erlernen.

In Kapitel 9, *Komplexe Selektoren und Durchlaufen des DOM*, erweitern Sie Ihre Kenntnisse über Selektoren und das Durchlaufen des DOM. Hier erwerben Sie die Fähigkeit, die Leistung von Selektoren zu optimieren, den DOM-Element-stack zu bearbeiten und Plug-ins zu schreiben, die die Auswahl- und Durchlaufmöglichkeiten erweitern.

In Kapitel 10, *Komplexe Ereignisse*, beschäftigen Sie sich eingehender mit Techniken wie Delegation und Drosselung, mit denen sich die Leistung der Ereignisbehandlung erheblich verbessern lässt. Außerdem erstellen Sie eigene und besondere Ereignisse, die die Möglichkeiten von jQuery noch erweitern.

In Kapitel 11, *Anspruchsvolle Effekte*, optimieren Sie die grafischen Effekte, die jQuery bietet, indem Sie benutzerdefinierte Easing-Funktionen erstellen und auf jeden Schritt einer Animation reagieren. Hier erhalten Sie die Möglichkeit, Animationen zu bearbeiten, während sie auftreten, und mit benutzerdefinierten Warteschleifen Aktionen nach Zeitplan ablaufen zu lassen.

In Kapitel 12, *DOM-Manipulation für Fortgeschrittene*, vertiefen Sie Ihre praktischen Kenntnisse in der Bearbeitung des DOM mit Techniken wie dem Anhängen beliebiger Daten an Elemente. Außerdem lernen Sie, wie Sie die Verarbeitung der CSS-Eigenschaften von Elementen durch jQuery erweitern.

Kapitel 13, *Ajax für Fortgeschrittene*, gibt Ihnen tiefere Einblicke in Ajax-Transaktionen, unter anderem in das jQuery-System für verzögerte Objekte, mit dem Daten gehandhabt werden, die erst später verfügbar werden.

In Anhang A, *JavaScript-Closures*, erhalten Sie solide Grundkenntnisse von Closures in JavaScript. Sie erfahren, worum es sich dabei handelt und wie Sie sie zu Ihrem Vorteil einsetzen können.

In Anhang B, *JavaScript mit QUnit testen*, lernen Sie die Bibliothek QUnit kennen, die für Unit-Tests von JavaScript-Programmen da ist. Diese Bibliothek ist eine wichtige Ergänzung Ihres Werkzeugkastens zur Entwicklung und Wartung anspruchsvoller Webanwendungen.

Anhang C, *Kurzreferenz*, gibt einen Überblick über die gesamte Bibliothek jQuery und führt dabei sämtliche Methoden und Selektorausdrücke auf. Die übersichtliche Gestaltung ist praktisch, wenn Sie wissen, was Sie tun wollen, aber nicht auf den richtigen Namen der gewünschten Methode bzw. des Selektors kommen.

## **Voraussetzungen für dieses Buch**

Um den Beispielcode auszuführen, der in diesem Buch vorgestellt wird, benötigen Sie einen modernen Webbrowser wie Mozilla Firefox, Apple Safari, Google Chrome oder Microsoft Internet Explorer.

Um mit den Beispiel herumzuspielen und die am Kapitelende aufgeführten Übungsaufgaben zu bearbeiten, benötigen Sie außerdem Folgendes:

- Einen einfachen Texteditor
- Webentwicklungstools für Ihren Browser wie z.B. Firebug (siehe Kapitel 1 im Abschnitt »Entwicklungswerzeuge«)
- Das komplette Codepaket für jedes Kapitel. Darin ist auch eine Kopie der Bibliothek jQuery enthalten (siehe den Abschnitt »Den Beispielcode herunterladen« weiter hinten).

Um einige der Ajax-Beispiele auszuführen, die ab Kapitel 6 vorgestellt werden, brauchen Sie einen Webserver mit aktiviertem PHP.

## Zielgruppe dieses Buches

Dieses Buch ist für Webdesigner gedacht, die interaktive Elemente erstellen möchten, und für Entwickler, die die bestmöglichen Benutzerschnittstellen für ihre Webanwendungen gestalten wollen. Grundkenntnisse in JavaScript-Programmierung werden vorausgesetzt. Sie müssen die Grundlagen von HTML und CSS kennen und sollten mit der Syntax von JavaScript vertraut sein. Vorkenntnisse in jQuery oder mit anderen JavaScript-Bibliotheken werden nicht benötigt.

Dieses Buch macht Sie mit dem Funktionsumfang und der Syntax von jQuery 1.6.x vertraut, der zurzeit neuesten Version.

## Die Geschichte des Projekts jQuery

Dieses Buch deckt den Funktionsumfang und die Syntax von jQuery 1.6.x ab, der zurzeit neuesten Version. Der Tenor der Bibliothek – eine einfache Möglichkeit zu bieten, um Elemente auf einer Webseite zu finden und sie zu bearbeiten – hat sich im Verlauf der Entwicklung nicht geändert, aber die Einzelheiten der Syntax und der Funktionsmerkmale. Diese kurze Übersicht über die Geschichte des Projekts beschreibt die wichtigsten Änderungen von einer Version zur nächsten, was für Leser, mit älteren Versionen der Bibliothek arbeiten, hilfreich sein mag.

- **Phase der öffentlichen Entwicklung:** Im August 2005 erwähnte John Resig erstmals eine Verbesserung der Prototype-Bibliothek Behavior. Dieses neue Framework wurde am 14. Januar 2006 formal als jQuery veröffentlicht.
- **jQuery 1.0 (August 2006):** Dieses erste stabile Release der Bibliothek bot bereits eine solide Unterstützung für CSS-Selektoren, Ereignisbehandlung und AJAX-Interaktion.
- **jQuery 1.1 (Januar 2007):** Mit diesem Release wurde die API erheblich verschlankt. Viele selten genutzte Methoden wurden kombiniert, sodass weniger Methoden zu lernen und zu dokumentieren waren.
- **jQuery 1.1.3 (Juli 2007):** Dieses untergeordnete Release bot erhebliche Geschwindigkeitssteigerungen der Selektor-Engine von jQuery. Seit dieser Option übertraf die Leistung von jQuery die anderer JavaScript-Bibliotheken wie Prototype, Mootools und Dojo.
- **jQuery 1.2 (September 2007):** In diesem Release wurde die XPath-Syntax für die Elementauswahl entfernt, da sie zur CSS-Syntax redundant geworden

war. Die Gestaltung von Effekten wurde in diesem Release flexibler, und die Entwicklung Plug-ins wurde durch die Ergänzung von Ereignis-Namensräumen vereinfacht.

- **jQuery UI** (September 2007): Die neue Plug-in-Suite wurde als Ersatz für das beliebte, aber veraltete Plug-in Interface angekündigt. Es enthielt eine reichhaltige Sammlung vorgefertigter Widgets sowie einen Satz von Werkzeugen zur Gestaltung anspruchsvoller Elemente wie Drag-&-Drop-Oberflächen.
- **jQuery 1.2.6** (Mai 2008): Die Funktionsmerkmale des beliebten Plug-ins Dimensions von Brandon Aaron wurden in die Hauptbibliothek aufgenommen.
- **jQuery 1.3** (Januar 2009): Eine Generalüberholung der Selektor-Engine (Sizzle) bot einen gewaltigen Schub für die Leistung der Bibliothek. Die Ereignisdelegierung wurde jetzt formal unterstützt.
- **jQuery 1.4** (Januar 2010): Diese Version, die wahrscheinlich ehrgeizigste Aktualisierung seit 1.0, brachte viele Leistungsverbesserungen für die DOM-Bearbeitung sowie eine große Menge neuer oder verbesserter Methoden für fast jeden Aspekt der Bibliothek. Das Erscheinen Version 1.4 wurde vierzehn Tage lang mit Ankündigungen und Videos auf einer eigens dazu angelegten Website (<http://jquery14.com/>) begleitet.
- **jQuery 1.4.2** (Februar 2010): Zwei neue Methoden zur Ereignisdelegierung wurden hinzugefügt (.delegate() und .undelegate()). Das gesamte Ereignissystem von jQuery wurde generalüberholt, um eine flexiblere Benutzung und eine größere browserübergreifende Konsistenz zu erreichen.
- **jQuery Mobile** (August 2010): Das jQuery-Projekt skizzierte öffentlich seine Strategie, seine Forschung und seine Benutzerschnittstellenentwürfe für die mobile Webelemente mit jQuery und einem neuen Mobilframework unter <http://jquerymobile.com/>.
- **jQuery 1.5** (Januar 2011): Die Ajax-Komponenten wurden erheblich umgeschrieben, um die Erweiterbarkeit und Leistung zu verbessern. Außerdem erhielt jQuery 1.5 eine Implementierung des Promise-Musters zur Handhabung von Abfragen von sowohl synchronen als auch asynchronen Funktionen.
- **jQuery 1.6** (Mai 2011): Die Komponente Attribute wurde umgeschrieben, um die Unterscheidung zwischen HTML-Attributen und DOM-Eigenschaften genauer widerzuspiegeln. Außerdem erhielt das in jQuery 1.5 eingeführte Objekt Deferred die beiden neuen Methoden .always() und .pipe().

### Historische Einzelheiten

Release Notes für ältere jQuery-Versionen sind auf der Projektwebsite unter <http://jquery.org/history> zu finden.

## Schreibweisen

In diesem Buch werden verschiedene Arten von Informationen durch unterschiedliche Formatierung des Textes gekennzeichnet. Im Folgenden finden Sie einige Beispiele für diese Formate und eine Erklärung ihrer Bedeutung.

Einzelne Codebegriffe werden im Text wie folgt dargestellt: »Dieser Code zeigt, dass wir der Methode `console.log()` jede Art von Ausdruck übergeben können.«

Ein ganzer Block von Code sieht folgendermaßen aus:

```
$('button.show-details').click(function() {  
    $('div.details').show();  
});
```

Um die Aufmerksamkeit auf einen bestimmten Teil eines Codeblocks zu lenken, sind die entsprechenden Zeilen oder Elemente fett hervorgehoben:

```
$('#switcher-narrow').bind('click', function() {  
    $('body').removeClass().addClass('narrow');  
});
```

*Neue Begriffe* und *wichtige Wörter* werden kursiv hervorgehoben. Begriffe, die Sie auf dem Bildschirm sehen, also z.B. in Menüs oder Dialogfeldern, werden folgendermaßen gekennzeichnet: »Die Registerkarte CONSOLE, die Sie im folgenden Screenshot sehen, werden wir am häufigsten verwenden, während wir den Umgang mit jQuery erlernen.«

Warnungen, wichtige Hinweise sowie Tipps und Tricks erscheinen in einem Kasten wie diesem.

## Herunterladen des Beispielcodes

Die Dateien mit dem Beispielcode können Sie für dieses Buch unter <http://www.dpunkt.de/jquery> herunterladen.

# 1 Erste Schritte

Das World Wide Web von heute stellt ein dynamisches Umfeld dar und seine Benutzer haben hohe Ansprüche sowohl an die Gestaltung als auch an die Funktion von Websites. Entwickler, die bemerkenswerte, interaktive Sites erstellen wollen, nutzen JavaScript-Bibliotheken wie jQuery, um häufig vorkommende Aufgaben zu automatisieren und komplizierte zu vereinfachen. Einer der Gründe für die Beliebtheit von jQuery besteht darin, dass diese Bibliothek bei einer breiten Palette von Aufgaben Hilfestellung geben kann.

Da jQuery so viele verschiedene Funktionen bietet, erscheint es schwierig, einen Anfang zu finden. Die Bibliothek ist jedoch logisch und konsistent aufgebaut, und viele ihrer Konzepte entstammen den Strukturen von HTML und CSS (Cascading Style Sheets). Die Gestaltung der Bibliothek ermöglicht auch Designern mit geringen Programmierkenntnissen einen schnellen Einstieg, da viele von ihnen mit diesen Technologien mehr Erfahrung haben als mit JavaScript. In diesem ersten Kapitel werden wir tatsächlich ein funktionierendes jQuery-Programm schreiben, das aus lediglich drei Zeilen Code besteht. Auch erfahrenen Programmierern hilft die konzeptuelle Klarheit, wie wir in späteren, anspruchsvolleren Kapiteln noch sehen werden.

Schauen wir uns also an, was jQuery für uns tun kann.

## 1.1 Was bietet jQuery?

Die Bibliothek jQuery bietet eine Allzweck-Abstraktionsschicht für die gängige Skripterstellung im Web und eignet sich daher für fast jede Situation, in der Sie Skripte einsetzen müssen. Da sie erweiterbar ist und ständig Plug-ins entwickelt werden, um neue Fähigkeiten hinzuzufügen, können wir niemals alle möglichen Verwendungsmöglichkeiten und Funktionen in einem Buch beschreiben. Die Kernfunktionen jedoch helfen bei der Erledigung der folgenden Aufgaben:

- Zugriff auf Elemente in einem Dokument: Ohne JavaScript-Bibliothek müssen Entwickler häufig viele Zeilen Code schreiben, um den *DOM*-Baum (*Document Object Model*) zu durchlaufen und einzelne Teile der Struktur

eines HTML-Dokuments zu finden. Mit jQuery steht den Entwicklern ein robuster und effizienter Selektionsmechanismus zur Verfügung, der es einfach macht, genau den Teil eines Dokuments abzurufen, den Sie untersuchen oder bearbeiten wollen.

```
$(‘div.content’).find(‘p’);
```

- Ändern des Erscheinungsbilds einer Webseite: CSS bietet sich als leistungsfähige Methode an, die Darstellung eines Dokuments zu beeinflussen, leidet jedoch darunter, dass Webbrowser die Standards unterschiedlich oder nur unvollständig unterstützen. Mit jQuery können Entwickler diese Lücke schließen und auf einer browserunabhängigen Abstraktion aufsetzen. Außerdem kann jQuery Klassen und einzelne Formateigenschaften, die einem Teil des Dokuments zugewiesen sind, auch dann noch ändern, wenn dieses bereits vom Browser dargestellt wird.

```
$(‘ul > li:first’).addClass(‘active’);
```

- Ändern des Dokumentinhalts: jQuery ist nicht auf die Modifikation der reinen Darstellung beschränkt, sondern kann auch den Inhalt eines Dokuments ändern, wozu nur wenige Tastendrücke erforderlich sind. So ist es möglich, Text zu ändern, Bilder einzufügen oder auszutauschen, Listen umzusortieren oder die gesamte HTML-Struktur umzustellen und zu erweitern, und das alles mit einer einzigen, einfach zu verwendenden *Anwendungsprogrammierschnittstelle (Application Programming Interface, API)*.

```
$(‘#container’).append(‘<a href=“more.html”>more</a>’);
```

- Reagieren auf die Aktionen des Benutzers: Selbst die raffiniertesten und leistungsfähigsten Verhaltensweisen nützen nichts, wenn wir nicht steuern können, wann sie stattfinden. Die Bibliothek jQuery bietet eine elegante Möglichkeit, ein breites Spektrum an Ereignissen, z.B. den Klick auf einen Link, abzufangen, ohne den HTML-Code dabei übermäßig mit deren Behandlung zu durchsetzen. Gleichzeitig überbrückt die API zur Ereignisbehandlung Browserunterschiede, die Webentwicklern häufig das Leben schwer machen.

```
$(‘button.show-details’).click(function() {  
    $(‘div.details’).show();  
});
```

- Animierte Darstellung von Änderungen an einem Dokument: Um solche interaktiven Änderungen wirkungsvoll umzusetzen, muss ein Designer dem Benutzer auch eine optische Rückmeldung geben. Die Bibliothek jQuery ermöglicht dies durch einen reichen Fundus an Effekten wie Fade (Ausblenden) und Wipe (Wegwischen) sowie durch einen Werkzeugkasten zum Gestalten eigener grafischer Effekte.

```
$(‘div.details’).slideDown();
```

## 1.2 Warum jQuery so gut funktioniert

- Abrufen von Informationen von einem Server, ohne die ganze Seite aktualisieren zu müssen: Dieses Codemuster ist als Ajax bekannt geworden, was ursprünglich für »asynchronous JavaScript and XML« stand, inzwischen aber eine viel breitere Palette von Technologien für die Kommunikation zwischen Client und Server umfasst. Die Bibliothek jQuery nimmt diesem aufwendigen und sehr vom darstellenden Endgerät abhängigen Vorgang die Komplexität browserspezifischer Aspekte, , sodass sich Entwickler auf die eigentliche Serverfunktionalität konzentrieren können.

```
$('div.details').load('more.html #content');
```

- Vereinfachen häufig vorkommender JavaScript-Aufgaben: Neben all den dokumentspezifischen Merkmalen bietet jQuery auch Erweiterungen zu grundlegenden JavaScript-Konstrukten wie Iteration und Array-Manipulation.

```
$.each(obj, function(key, value) {  
    total += value;  
});
```

### Den Beispielcode herunterladen

Die Dateien mit dem Beispielcode können Sie unter <http://www.dpunkt.de/jquery> herunterladen.

## 1.2 Warum jQuery so gut funktioniert

Mit dem Wiederaufleben des Interesses an dynamischem HTML kam es zu einer starken Vermehrung der JavaScript-Frameworks. Manche sind spezialisiert und konzentrieren sich nur auf ein oder zwei Aufgaben, während andere versuchen, alle möglichen Verhaltensweisen und Animationen zu katalogisieren und sie vorgefertigt anzubieten. Um das oben angedeutete breite Spektrum an Funktionen zu bieten und gleichzeitig relativ kompakt zu bleiben, werden bei jQuery die folgenden Strategien eingesetzt:

- Nutzen von CSS-Kenntnissen: Indem der Mechanismus zum Auffinden von Seitenelementen auf CSS-Selektoren basiert, erbt jQuery eine knappe und doch lesbare Art, um die Struktur eines Dokuments auszudrücken. Die Bibliothek jQuery ist ein Einstiegspunkt für Designer, die ihren Seiten Verhaltensweisen hinzufügen möchten, da Kenntnisse der CSS-Syntax eine Voraussetzung für die professionelle Webentwicklung sind.
- Unterstützung von Erweiterungen: Um einen schleichenden Funktionszuwachs zu vermeiden, werden bei jQuery Sonderfälle in *Plug-ins* verbannt. Die Methode zum Erstellen neuer Plug-ins ist einfach und gut dokumentiert, was die Entwicklung einer großen Vielfalt origineller und nützlicher Module gefördert hat. Die meisten Funktionen im grundlegenden jQuery-Download sind intern durch die Plug-in-Architektur realisiert und können falls gewünscht entfernt werden, was zu einer noch kleineren Bibliothek führt.

- Wegabstrahieren von Eigenheiten der Browser: Es ist eine bedauernswerte Tatsache in der Webentwicklung, dass jeder Browser seine eigenen Bereiche hat, in denen er von den öffentlichen Standards abweicht. Ein erheblicher Anteil jeder Webanwendung kann ausgelagert werden, um sich um die Merkmale zu kümmern, die auf jeder Plattform anders sind. Während die im ständigen Wandel begriffene Browserlandschaft es unmöglich macht, für bestimmte anspruchsvolle Funktionen vollständig browserneutralen Code zu schreiben, fügt jQuery eine *Abstraktionsschicht* hinzu, die häufig vorkommende Aufgaben normalisiert, den Code erheblich vereinfacht und seinen Umfang verringert.
- Konsequente Verwendung von Mengen: Wenn wir jQuery anweisen: »Finde alle Elemente der Klasse `collapsible` und blende sie aus (»hide them«)«, besteht keine Notwendigkeit, jedes zurückgegebene Element in einer Schleife durchzugehen. Stattdessen sind Methoden wie `.hide()` dazu da, automatisch Mengen von Objekten zu verarbeiten statt Einzelobjekte. Diese Technik, die *implizite Iteration*, bedeutet, dass viele Schleifenkonstrukte unnötig werden, was den Code erheblich verkürzt.
- Zulassen mehrerer Aktionen in einer Zeile: Um die übermäßige Verwendung temporärer Variablen und verschwenderische Wiederholungen zu vermeiden, nutzt jQuery für die meisten seiner Methoden das Programmiermuster der *Verkettung*. Das bedeutet, dass das Ergebnis der meisten Operationen an einem Objekt das Objekt selbst ist, das gleich wieder für die nächste Aktion bereitsteht.

Diese Strategien haben dafür gesorgt, dass das jQuery-Paket schlank bleibt – komprimiert ungefähr 30 KB –, während es gleichzeitig Techniken bereitstellt, die unseren selbst geschriebenen Code, der die Bibliothek nutzt, kompakt halten.

Die Eleganz dieser Bibliothek gründet sich zum Teil auf das Design, zum Teil auf den evolutionären Prozess, den die rund um das Projekt aufgekommene, lebendige Community fördert. Benutzer von jQuery diskutieren nicht nur die Entwicklung von Plug-ins, sondern auch Verbesserungen an der Kernbibliothek. Außerdem tragen sowohl Benutzer als auch Entwickler zu der ständigen Verbesserung der offiziellen Projektdokumentation bei, die Sie unter <http://api.jquery.com> finden können.

Obwohl zur Konstruktion eines so flexiblen und stabilen Systems viel Arbeit erforderlich ist, kann das Endprodukt kostenlos genutzt werden. Dieses Open-Source-Projekt steht unter den beiden Lizenzen *MIT Licence* (zur freien Verwendung von jQuery auf jeder Site und zu seiner Nutzung in proprietärer Software) und *GNU Public Licence* (für die Einbindung in andere Open-Source-Projekte mit GNU-Lizenz) zur Verfügung.

## 1.3 Unsere erste Webseite mit jQuery

Nachdem wir uns damit befasst haben, welch ein Umfang an Funktionalität in jQuery zur Verfügung steht, sehen wir uns nun an, wie wir die Bibliothek tatsächlich nutzen. Dazu benötigen wir zunächst eine Kopie von jQuery.

### 1.3.1 jQuery herunterladen

Eine Installation ist nicht erforderlich. Um jQuery verwenden zu können, brauchen wir lediglich eine öffentlich verfügbare Kopie der Datei, die sich auf einer externen Site oder auf unserer eigenen befinden kann. Da JavaScript eine interpretierte Sprache ist, gibt es keine Kompilierung und keine Build-Phase, über die Sie sich Sorgen machen müssten. Wann immer wir auf einer Seite jQuery zur Verfügung haben müssen, verweisen wir einfach in einem `<script>`-Element des HTML-Dokuments auf den Speicherort der Datei.

Die offizielle jQuery-Website (<http://jquery.com/>) beinhaltet stets die aktuellste stabile Version der Bibliothek, die Sie gleich von der Homepage herunterladen können. Es kann sein, dass gleichzeitig mehrere Versionen von jQuery verfügbar sind. Für uns als Websiteentwickler ist die neueste unkomprimierte Version geeignet. In Produktionsumgebungen kann sie durch eine komprimierte Version ersetzt werden.

Angesichts der steigenden Beliebtheit von jQuery haben manche Unternehmen die Datei in ihren *CDNs* (*Content Delivery Networks*) kostenlos zur Verfügung gestellt. Vor allem Google (<http://code.google.com/apis/ajaxlibs/documentation/>) und Microsoft (<http://www.asp.net/ajax/cdn>) bieten die Datei auf leistungsfähigen, verzögerungssarmen Servern überall auf der Welt an, um einen schnellen Download unabhängig vom Standort des Benutzers sicherzustellen. Die CDN-Kopien von jQuery bieten aufgrund der Serververteilung und Zwischenspeicherung zwar Geschwindigkeitsvorteile, doch kann während der Entwicklung die Verwendung einer lokalen Kopie praktischer sein. In den Beispielen in diesem Buch nutzen wir eine Kopie der Datei in unserem eigenen System, sodass wir den Code ausführen können, unabhängig davon, ob wir mit dem Internet verbunden sind oder nicht.

### 1.3.2 Einrichten von jQuery in einem HTML-Dokument

Die meisten Beispiele für die Nutzung von jQuery bestehen aus drei Teilen: dem HTML-Dokument, der CSS-Datei zu seiner Formatierung und den JavaScript-Dateien, die die Interaktionen definieren. Als erstes Beispiel verwenden wir eine Seite mit dem Auszug aus einem Buch, dessen einzelnen Teilen eine Reihe von Klassen zugewiesen sind. Diese Seite enthält einen Verweis auf die neueste Version der Bibliothek jQuery, die wir heruntergeladen, in `jquery.js` umbenannt und in unserem lokalen Projektverzeichnis platziert haben. Das HTML-Dokument sieht wie folgt aus:

```
<!DOCTYPE html>

<html lang="en">
  <head>
    <meta charset="utf-8">
    <title>Through the Looking-Glass</title>
    <link rel="stylesheet" href="01.css">
    <script src="jquery.js"></script>
    <script src="01.js"></script>
  </head>

  <body>
    <h1>Through the Looking-Glass</h1>
    <div class="author">by Lewis Carroll</div>

    <div class="chapter" id="chapter-1">
      <h2 class="chapter-title">1. Looking-Glass House</h2>
      <p>There was a book lying near Alice on the table,  

         and while she sat watching the White King (for she  

         was still a little anxious about him, and had the  

         ink all ready to throw over him, in case he fainted  

         again), she turned over the leaves, to find some  

         part that she could read,  

         <span class="spoken">  

           "&mdash;for it's all in some language I don't know,"  

         </span>  

         she said to herself.  

      </p>
      <p>It was like this.</p>
      <div class="poem">
        <h3 class="poem-title">YKCOWREBBAJ</h3>
        <div class="poem-stanza">
          <div>sevot yhtils eht dna ,gillirb sawI'</div>
          <div>;ebaw eht ni elbmig dna eryg did</div>
          <div>,sevogorob eht erew ysmin lla</div>
          <div>.ebargtuo shtar emom eht dnA</div>
        </div>
      </div>
      <p>She puzzled over this for some time, but at last  

         a bright thought struck her.  

         <span class="spoken">  

           "Why, it's a Looking-glass book, of course! And if  

           I hold it up to a glass, the words will all go the  

           right way again."  

         </span>
      </p>
      <p>This was the poem that Alice read.</p>
      <div class="poem">
        <h3 class="poem-title">JABBERWOCKY</h3>
        <div class="poem-stanza">
          <div>'Twas brillig, and the slithy toves</div>
```

```
<div>Did gyre and gimble in the wabe;</div>
<div>All mimsy were the borogoves,</div>
<div>And the mome raths outgrabe.</div>
</div>
</div>
</div>
</body>
</html>
```

### Dateipfade

Das tatsächliche Layout der Dateien auf dem Server spielt keine Rolle. Verweise von einer Datei auf eine andere müssen nur auf die gewählte Struktur angepasst werden. In den meisten Beispielen in diesem Buch verwenden wir für Verweise auf Dateien relative Pfade (../images/foo.png) und keine absoluten (/images/foo.png). Dadurch kann der Code lokal ausgeführt werden, ohne dass ein Webserver nötig ist.

Unmittelbar nach der normalen HTML-Präambel wird das Stylesheet geladen. Für dieses Beispiel verwenden wir das folgende:

```
body {
    background-color: #fff;
    color: #000;
    font-family: Helvetica, Arial, sans-serif;
}
h1, h2, h3 {
    margin-bottom: .2em;
}
.poem {
    margin: 0 2em;
}
.highlight {
    background-color: #ccc;
    border: 1px solid #888;
    font-style: italic;
    margin: 0.5em 0;
    padding: 0.5em;
}
```

Nach dem Verweis auf das Stylesheet werden die JavaScript-Dateien eingeschlossen. Es ist wichtig, dass sich das `<script>`-Tag für die Bibliothek jQuery vor den Tags für unsere eigenen Skripte befindet, da das jQuery-Framework sonst nicht verfügbar wäre, wenn unser Code versucht, darauf zu verweisen.

Im weiteren Verlauf dieses Buches werden immer nur die jeweils relevanten Teile der HTML- und CSS-Dateien gezeigt. Die vollständigen Dateien stehen auf der Begleitwebsite zu diesem Buch unter <http://www.dpunkt.de/jquery> zur Verfügung.

Die entsprechende Seite ist in folgender Abbildung dargestellt:

## Through the Looking-Glass

by Lewis Carroll

### 1. Looking-Glass House

There was a book lying near Alice on the table, and while she sat watching the White King (for she was still a little anxious about him, and had the ink all ready to throw over him, in case he fainted again), she turned over the leaves, to find some part that she could read, "—for it's all in some language I don't know," she said to herself.

It was like this.

#### YKCOWREBBAJ

sevot yhtils eht dna „gillirb sawT“  
;ebaw eht ni elbmig dna eryg diD  
.sevogorob eht erew ysmin IIA  
.ebargtuo sthtar emom eht dnA

She puzzled over this for some time, but at last a bright thought struck her. "Why, it's a Looking-glass book, of course! And if I hold it up to a glass, the words will all go the right way again."

This was the poem that Alice read.

#### JABBERWOCKY

'Twas brillig, and the slithy toves  
Did gyre and gimble in the wabe;  
All mimsy were the borogoves,  
And the mome raths outgrabe.

Als Nächstes verwenden wir jQuery, um dem Gedicht ein neues Layoutformat zuzuweisen.

Dieses Beispiel soll zeigen, wie einfach jQuery eingesetzt werden kann. In der Praxis führen Sie eine solche Formatierung ausschließlich mithilfe von CSS durch.

### 1.3.3 jQuery-Code hinzufügen

Unseren eigenen Code platzieren wir in der zweiten, zurzeit leeren JavaScript-Datei, die wir mit `<script src="01.js"></script>` in das HTML-Dokument eingefügt haben. In diesem Beispiel brauchen wir nur drei Zeilen Code:

```
$(document).ready(function() {  
    $('div.poem-stanza').addClass('highlight');  
});
```

### Das Gedicht finden

Die grundlegende Operation in jQuery besteht darin, einen Teil des Dokuments auszuwählen. Dies geschieht mit der Funktion `$()`. Gewöhnlich erhält sie als Parameter einen String, der einen beliebigen CSS-Selektorausdruck enthalten

kann. In unserem Beispiel möchten wir alle <div>-Elemente in dem Dokument finden, denen die Klasse poem-stanza zugewiesen ist, weshalb der Selektor sehr einfach ist. Im weiteren Verlauf dieses Buches werden wir uns jedoch noch weit anspruchsvollere Optionen ansehen. In Kapitel 2, »Elemente auswählen«, werden wir viele Möglichkeiten kennenlernen, um einzelne Teile eines Dokuments ausfindig zu machen.

Wenn die Funktion `$( )` aufgerufen wird, gibt sie eine neue *jQuery-Objektinstanz* zurück. Das ist der Grundbaustein, mit dem wir von nun an arbeiten werden. Dieses Objekt kann null oder mehrere DOM-Elemente kapseln und erlaubt uns, mit ihnen auf verschiedene Weise umzugehen. In unserem Fall wollen wir das Erscheinungsbild dieser Abschnitte auf der Seite ändern. Das erreichen wir, indem wir dem Gedichttext andere Klassen zuweisen.

### Eine neue Klasse einfügen

Wie die meisten jQuery-Methoden trägt auch `.addClass()` einen Namen, der genau ihren Zweck angibt: Sie wendet eine CSS-Klasse auf den Teil der Seite an, den wir ausgewählt haben. Ihr einziger Parameter ist der Name der hinzuzufügenden Klasse. Mithilfe dieser Methode und ihres Gegenstücks `.removeClass()` können wir jQuery in Aktion beobachten, während wir die verschiedenen Selektorausdrücke ausprobieren. Vorerst fügen wir in unserem Beispiel einfach die Klasse `highlight` hinzu, die in unserem Stylesheet als kursiver Text mit grauem Hintergrund und Rand definiert ist.

Beachten Sie, dass keine Iteration notwendig ist, um die Klasse allen Gedichtstrophen hinzuzufügen. Wie wir bereits gesagt haben, führt jQuery in Methoden wie `.addClass()` eine *implizite Iteration* durch, sodass ein einziger Funktionsaufruf ausreicht, um alle ausgewählten Teile des Dokuments zu ändern.

### Den Code ausführen

Die Kombination von `$( )` und `.addClass()` reicht aus, um unser Ziel zu erreichen, das Erscheinungsbild des Gedichttextes zu ändern. Wenn wir diese Codezeile jedoch einfach in den Dokument-Header einfügen, wird sie keine Auswirkung zeigen. JavaScript-Code wird nämlich im Allgemeinen ausgeführt, sobald der Browser darauf stößt, und während der Verarbeitung des Headers ist noch gar kein HTML-Code vorhanden, der formatiert werden könnte. Wir müssen die Ausführung des Codes daher verzögern, bis das DOM für uns bereitsteht.

Mit dem Konstrukt `$(document).ready()` erlaubt uns jQuery, das Auslösen von Funktionsaufrufen für den Zeitpunkt einzuplanen, an dem das DOM geladen ist, wobei nicht unbedingt darauf gewartet wird, dass alle Bilder vollständig dargestellt werden. Eine solche zeitliche Planung von Ereignissen ist zwar auch ohne jQuery möglich, doch `$(document).ready()` bildet eine besonders elegante, browserübergreifende Lösung mit diversen Vorteilen:

- Sie nutzt die browsereigene Implementierung des Ereignisses DOM ready, wenn sie verfügbar ist, und fügt als Sicherheitsnetz noch den Ereignishandler window.onload hinzu.
- Sie ermöglicht mehrere Aufrufe von \$(document).ready() und führt sie in der Reihenfolge der Aufrufe durch.
- Sie führt an \$(document).ready() übergebene Funktionen selbst dann aus, wenn sie hinzugefügt werden, nachdem das Browserereignis bereits aufgetreten ist.
- Sie führt die Ereignisplanung asynchron durch, damit Skripte bei Bedarf für eine Verzögerung sorgen können.
- Sie simuliert in einigen Browsern ein DOM-ready-Ereignis, indem sie wiederholt nach dem Vorhandensein einer DOM-Methode sucht, die normalerweise zur selben Zeit verfügbar wird wie das DOM.

Die Parameter der Methode .ready() können einen Verweis auf eine bereits definierte Funktion übernehmen, wie der folgende Codeausschnitt zeigt:

```
function addHighlightClass() {  
    $('div.poem-stanza').addClass('highlight');  
}  
  
$(document).ready(addHighlightClass);
```

**Listing 1–1**

Wie die ursprüngliche Version des Skripts und das folgende Listing 1–2 zeigen, akzeptiert die Methode auch eine *anonyme Funktion* (manchmal auch *Lambda-Funktion* genannt):

```
$(document).ready(function() {  
    $('div.poem-stanza').addClass('highlight');  
});
```

**Listing 1–2**

Diese Ausdrucksweise mit anonymer Funktion ist in jQuery sehr praktisch für Methoden, die eine nicht wiederverwendbare Funktion als Argument übernehmen. Außerdem kann der dadurch erzeugte *Funktionseinschluss* (auch *Hülle* oder *Closure* genannt) ein anspruchsvolles und leistungsfähiges Werkzeug sein. Allerdings können sich daraus auch unerwünschte Folgen für die Speichernutzung ergeben, wenn man nicht vorsichtig ist. Funktionseinschlüsse werden ausführlich in Anhang A, »*JavaScript-Closures*«, behandelt.

### 1.3.4 Das fertige Produkt

Mit unserem JavaScript-Code sieht unsere Seite jetzt wie im folgenden Screenshot aus:

# Through the Looking-Glass

by Lewis Carroll

## 1. Looking-Glass House

There was a book lying near Alice on the table, and while she sat watching the White King (for she was still a little anxious about him, and had the ink all ready to throw over him, in case he fainted again), she turned over the leaves, to find some part that she could read, “—for it’s all in some language I don’t know,” she said to herself.

It was like this.

### YKCOWREBBAJ

```
sevot yhtils eht dna ,gillrb sawT'  
;ebaw eht ni elbmig dna eryg dID  
,sevogorob eht erek ysmlm lIA  
.ebargtuo shtar emom eht dnA
```

She puzzled over this for some time, but at last a bright thought struck her. “Why, it’s a Looking-glass book, of course! And if I hold it up to a glass, the words will all go the right way again.”

This was the poem that Alice read.

### JABBERWOCKY

```
'Twas brillig, and the slithy toves  
Did gyre and gimble in the wabe;  
All mimsy were the borogoves,  
And the mome raths outgrabe.
```

Da der JavaScript-Code die Klasse `highlight` eingefügt hat, erscheinen die Gedichtstrophen jetzt wie im Stylesheet 01.css definiert, also kursiv und in Kästen mit grauem Hintergrund.

## 1.4 Einfaches JavaScript und jQuery im Vergleich

Ohne jQuery kann selbst eine so einfache Aufgabe wie diese kompliziert sein. In einfachem JavaScript hätten wir die Klasse `highlight` wie im folgenden Codeausschnitt hinzufügen können:

```
window.onload = function() {  
    var divs = document.getElementsByTagName('div');  
    for (var i = 0; i < divs.length; i++) {  
        if (hasClass(divs[i], 'poem-stanza')  
            && !hasClass(divs[i], 'highlight')) {  
            divs[i].className += ' highlight';  
        }  
    }  
  
    function hasClass( elem, cls ) {  
        var reClass = new RegExp(' ' + cls + ' ');  
        return reClass.test(' ' + elem.className + ' ');  
    }  
};
```

**Listing 1–3**

Trotz ihrer Länge kann diese Lösung viele der Aufgaben, um die sich jQuery in Listing 1–2 für uns kümmert, nicht erledigen, wie z.B. die folgenden:

- Korrekte Berücksichtigung anderer `window.onload`-Ereignishandler
- Sofortiges Handeln nach der Bereitstellung des DOM
- Optimieren des Elementabrufs und anderer Aufgaben mit modernen DOM-Methoden

Wie Sie sehen, lässt sich jQuery-Code einfacher schreiben, einfacher lesen und schneller ausführen als das Gegenstück in einfachem JavaScript.

## 1.5 Entwicklungswerkzeuge

Wie dieser Codevergleich zeigt, ist jQuery-Code gewöhnlich kürzer und klarer als das Gegenstück in einfachem JavaScript. Das heißt jedoch nicht, dass wir stets fehlerfreien Code schreiben oder jederzeit intuitiv verstehen können, was auf unseren Seiten geschieht. Das Schreiben von jQuery-Code wird uns viel leichter von der Hand gehen, wenn wir gängige Entwicklungswerkzeuge zu Hilfe nehmen.

In allen modernen Browsern stehen hochwertige Entwicklungswerkzeuge zur Verfügung, weshalb wir einfach die Umgebung wählen können, in der wir uns am meisten zu Hause fühlen. Unter anderem bieten sich uns folgende Möglichkeiten:

- **Internet Explorer-Entwicklertools:**

*<http://msdn.microsoft.com/de-de/library/dd565628.aspx>*

- **Safari Web Inspector:**

*<http://developer.apple.com/technologies/safari/developer-tools.html>*

- **Chrome Developer Tools:**

*<http://code.google.com/intl/de-DE/chrome/devtools/>*

- **Firebug für Firefox:**

*<http://getfirebug.com/>*

Jedes dieser Instrumente bietet ähnliche Entwicklungsfunktionen:

- Die Möglichkeit, Aspekte des DOM zu untersuchen und zu ändern
- Die Möglichkeit, die Beziehungen zwischen dem CSS-Code und seinen Auswirkungen auf die Seitendarstellung zu untersuchen
- Bequeme Verfolgung der Skriptausführung durch spezielle Methoden
- Anhalten der Ausführung laufender Skripte und Untersuchen von Variablenwerten

In den Einzelheiten unterscheiden sich diese Funktionen zwar von Browser zu Browser, aber das Grundprinzip bleibt das gleiche. In diesem Buch erfordern

einige Beispiele die Verwendung eines dieser Werkzeuge. Zur Veranschaulichung setzen wir hier Firebug ein, aber die Entwicklungswerkzeuge für die anderen Browser sind ebenfalls gute Alternativen.

### 1.5.1 Firebug

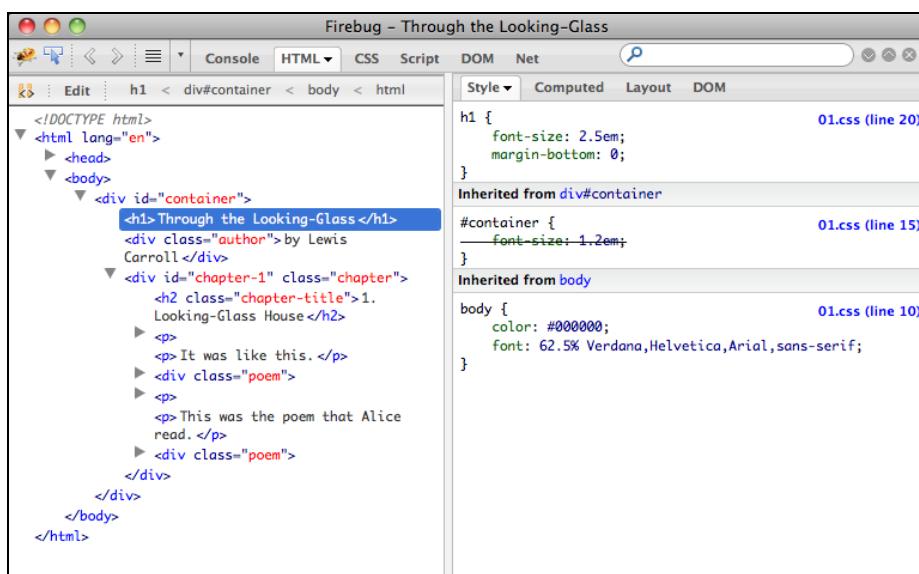
Aktuelle Anleitungen zur Installation und Verwendung von Firebug finden Sie auf der Projekthomepage unter <http://getfirebug.com/>. Dieses Werkzeug ist zu vielschichtig, um es hier ausführlich zu beschreiben. Stattdessen geben wir nur einen Überblick über einige der wichtigsten Funktionen.

#### Anmerkung zu den Screenshots

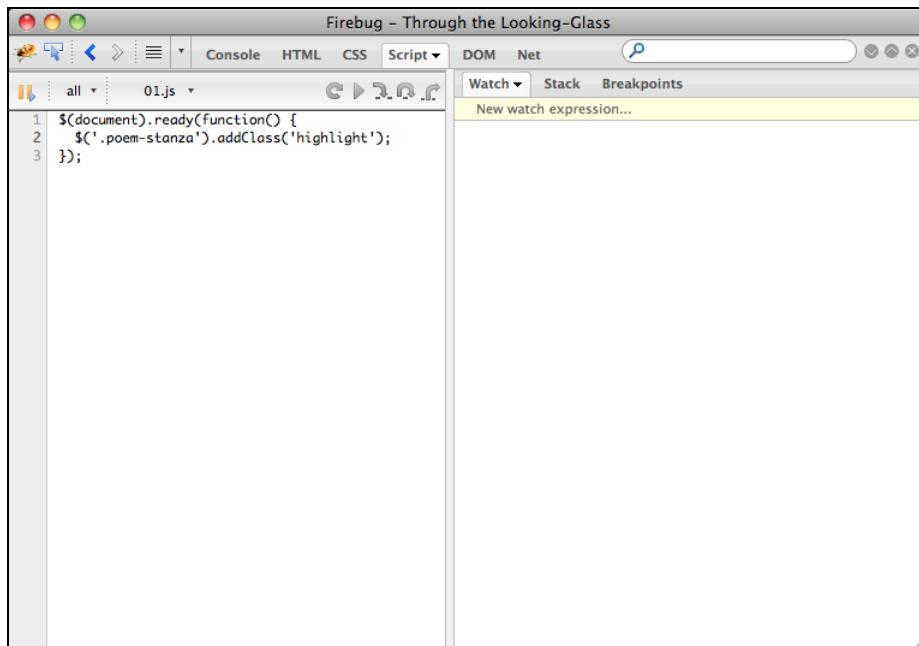
Firebug wird rasch weiterentwickelt, weshalb die folgenden Screenshots von dem abweichen können, was Sie in Ihrer Umgebung zu sehen bekommen. Einige der Beschriftungen und Schaltflächen kommen durch das optionale Add-on *FireQuery* zustande, das Sie unter <http://firequery.binaryage.com/> erhalten.

Wenn Firebug aktiviert ist, wird ein neues Bedienfeld mit Informationen über die aktuelle Seite angezeigt.

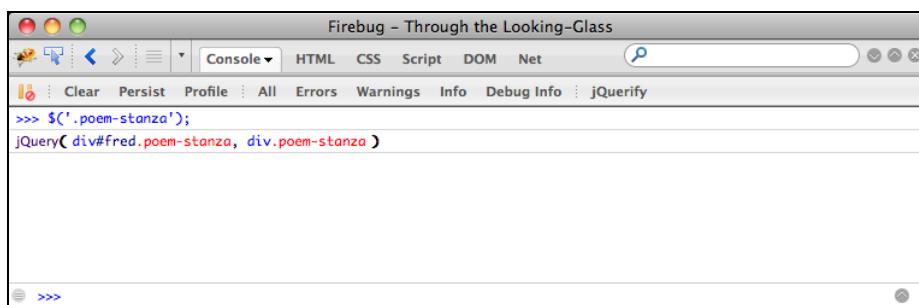
Auf der Standardregisterkarte HTML dieses Bedienfelds ist auf der linken Seite eine Darstellung der Seitenstruktur zu sehen und auf der rechten Seite Angaben über das jeweils markierte Element (z.B. die CSS-Regeln, die darauf angewandt werden). Diese Registerkarte eignet sich vor allem zur Untersuchung der Seitenstruktur und zur Behebung von CSS-Fehlern, wie Sie im folgenden Screenshot erkennen können:



Auf der Registerkarte SCRIPT können wir uns den Inhalt aller geladenen Skripte auf der Seite ansehen, wie der folgende Screenshot zeigt. Durch einen Klick auf eine Zeilennummer setzen wir einen Haltepunkt (breakpoint). Wenn das Skript eine Zeile mit einem solchen Haltepunkt erreicht, wird seine Ausführung angehalten, bis wir sie mit einem Klick auf eine Schaltfläche wieder aufnehmen. Auf der rechten Seite können wir Variablen und Ausdrücke eingeben, deren Wert wir jederzeit ablesen möchten.



Beim Erlernen von jQuery werden wir am häufigsten die Registerkarte CONSOLE verwenden, die Sie im folgenden Screenshot sehen. In einem Feld am unteren Rand können wir eine JavaScript-Anweisung eingeben, deren Ergebnis dann in dem Bedienfeld angezeigt wird:



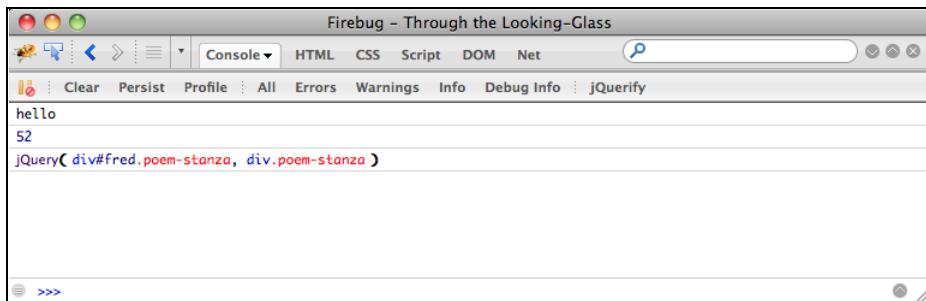
In diesem Beispiel haben wir denselben jQuery-Selektor verwendet wie in Listing 1–2, aber keine Aktion an den ausgewählten Elementen durchgeführt. Doch selbst in dieser Form gibt uns die Anweisung interessante Informationen. Wir können erkennen, dass das Ergebnis des Selektors ein jQuery-Objekt ist, das auf zwei `.poem-stanza`-Elemente auf der Seite verweist. Mit dieser Konsolenfunktion können wir jederzeit jQuery-Code schnell ausprobieren, und zwar direkt im Browser.

Außerdem können wir auch direkt in unserem Code mit der Konsole in Verbindung treten, indem wir wie folgt die Methode `console.log()` verwenden:

```
$(document).ready(function() {  
    console.log('hello');  
    console.log(52);  
    console.log($('div.poem-stanza'));  
});
```

**Listing 1–4**

Dieser Code zeigt, dass wir der Methode `console.log()` jede Art von Ausdruck übergeben können. Einfache Werte wie Strings und Zahlen werden unmittelbar ausgegeben, kompliziertere Werte wie jQuery-Objekte dagegen werden, wie der folgende Screenshot zeigt, zur Inspektion übersichtlich dargestellt:



Die Funktion `console.log()` (die in jedem der erwähnten Browser-Entwicklungswerkzeuge funktioniert) ist eine zweckmäßige Alternative zur JavaScript-Funktion `alert()`. Sie wird sich vor allem beim Testen unseres jQuery-Codes als sehr praktisch erweisen.

## 1.6 Zusammenfassung

Wir haben jetzt eine Vorstellung davon, warum ein Entwickler selbst für die einfachsten Aufgaben lieber ein JavaScript-Framework verwendet, als den gesamten Code von Grund auf selbst zu schreiben. Außerdem haben wir gesehen, in welchen Punkten sich jQuery als Framework besonders auszeichnet, warum wir es gegenüber anderen Möglichkeiten bevorzugen und welche allgemeinen Aufgaben es erleichtert.

In diesem Kapitel haben wir gelernt, wie wir jQuery für den JavaScript-Code auf unserer Website verfügbar machen. Wir wissen nun, wie wir mit der Funktion `$( )` nach einem Bestandteil der Seite suchen, der einer bestimmten Klasse zugeordnet ist, wie wir mit `.addClass()` eine zusätzliche Formatierung auf diesen Teil der Seite anwenden und wie wir mit `$(document).ready()` dafür sorgen, dass der Code erst nach dem Laden der Seite ausgeführt wird. Darüber hinaus haben wir uns die Entwicklungswerkzeuge angesehen, die wir beim Schreiben, Testen und Korrigieren unseres jQuery-Codes einsetzen werden.

Das einfache Beispiel, das wir hier verwendet haben, zeigt zwar gut, wie jQuery funktioniert, ist in der Praxis aber nicht sehr sinnvoll. Im nächsten Kapitel erweitern wir den Code, indem wir die anspruchsvolle Selektorensprache von jQuery erlernen, um praktische Anwendungen für diese Technik zu finden.

## 2 Elemente auswählen

Die Bibliothek jQuery macht die umfassenden Möglichkeiten von CSS-*Selektoren* nutzbar, damit wir auf schnelle und einfache Weise auf Elemente oder Gruppen von Elementen im Document Object Model (DOM) zugreifen können. In diesem Kapitel sehen wir uns einige dieser Selektoren sowie die eigenen Selektoren von jQuery an. Außerdem beschäftigen wir uns mit den *Methoden zur Traversierung des DOM*, die uns noch mehr Möglichkeiten bieten, um das zu bekommen, was wir wollen.

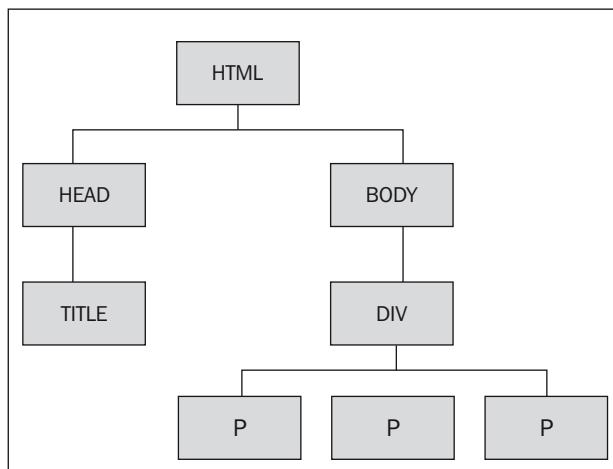
### 2.1 Das Document Object Model

Einer der leistungsfähigsten Aspekte von jQuery besteht darin, die Auswahl von Elementen im DOM zu vereinfachen. Das Document Object Model dient als Schnittstelle zwischen JavaScript und einer Webseite und bietet eine Repräsentation des HTML-Quelltextes als Verbund von Objekten statt als reiner Text.

Dieser Verbund nimmt die Form eines »Familienstammbaums« der Elemente auf der Seite an. Wenn wir über die Beziehungen zwischen den Elementen sprechen, verwenden wir dieselben Begriffe wie für Verwandtschaftsverhältnisse: Eltern, Kinder usw. Das folgende einfache Beispiel zeigt, wie das Bild des Familienstammbaums auf ein Dokument angewandt werden kann:

```
<html>
  <head>
    <title>the title</title>
  </head>
  <body>
    <div>
      <p>This is a paragraph.</p>
      <p>This is another paragraph.</p>
      <p>This is yet another paragraph.</p>
    </div>
  </body>
</html>
```

Hier ist `<html>` der Vorfahr aller anderen Elemente – mit anderen Worten, alle anderen Elemente sind Nachfahren von `<html>`. Die Elemente `<head>` und `<body>` sind nicht nur allgemein Nachfahren, sondern Kinder von `<html>`, und ebenso ist `<html>` nicht nur ihr Vorfahr, sondern ihr Elternelement. Die `<p>`-Elemente sind Kinder (und damit natürlich nach Nachfahren) von `<div>`, Nachfahren von `<body>` und `<html>` sowie untereinander Geschwister. Das alles können Sie an folgendem Diagramm ablesen:



Um die Stammbaumstruktur des DOM grafisch darzustellen, können Sie eine Reihe von Softwarewerkzeugen verwenden, z.B. das Firefox-Plug-in Firebug oder den Web Inspector für Safari und Chrome.

Da uns dieser Elementbaum zur Verfügung steht, können wir jQuery einsetzen, um beliebige Mengen von Elementen auf der Seite zu finden. Unsere Werkzeuge dafür sind die jQuery-Selektoren und die Methoden zur Traversierung des DOM-Baums.

Bevor wir fortfahren, müssen wir jedoch noch einen wichtigen Punkt festhalten: Die aus der Verwendung der Selektoren und Methoden resultierende Menge der Elemente ist stets in ein *jQuery-Objekt* eingehüllt. Mit diesen Objekten können wir sehr einfach arbeiten, wenn wir etwas mit den Dingen tun wollen, die wir auf der Seite finden. So können wir sehr einfach *Ereignisse* an diese Objekte binden, ansprechende *Effekte* auf sie anwenden und sogar mehrere Änderungen oder Effekte *verketten*. Dennoch unterscheiden sich jQuery-Objekte von regulären DOM-Elementen oder Knotenlisten, da sie für einige Aufgabe nicht unbedingt dieselben Methoden und Eigenschaften bereitstellen. Im letzten Teil dieses Kapitels sehen wir uns daher eine Möglichkeit an, um unmittelbar auf die in einem jQuery-Objekt verpackten DOM-Elemente zuzugreifen.

## 2.2 Die Funktion `$()`

Unabhängig davon, welche Art von Selektor wir in jQuery verwenden möchten, beginnen wir stets mit derselben Funktion, nämlich mit `$()`. Sie akzeptiert im Normalfall einen CSS-Selektor als einzigen Parameter und dient dazu, ein neues jQuery-Objekt zurückzugeben, das auf das entsprechende Element auf der Seite zeigt. Praktisch alles, was Sie in einem Stylesheet verwenden können, lässt sich auch als String an diese Funktion übergeben, sodass wir jQuery-Methoden auf die Menge der passenden Elemente anwenden können.

### jQuery mit anderen JavaScript-Bibliotheken in Einklang bringen

In jQuery ist das Dollarzeichen einfach ein »Alias« für jQuery. Da die Funktion `$()` in JavaScript-Bibliotheken häufig vorkommt, können Konflikte entstehen, wenn auf einer Seite mehrere dieser Bibliotheken verwendet werden. Um solche Konflikte zu vermeiden, könnten wir jedes `$` in unserem jQuery-Code durch `jQuery` ersetzen. Weitere Lösungen für dieses Problem werden in Kapitel 10, »Komplexe Ereignisse«, ange- sprochen.

Die drei primären Elemente, aus denen Selektoren aufgebaut sind, sind Tag-Name, ID und Klasse. Diese können einzeln oder in Kombination miteinander verwendet werden. Die folgenden einfachen Beispiele zeigen, wie diese Selektoren im Code dargestellt werden:

Selektor-Typ	CSS	jQuery	Wirkung
Tag-Name	<code>p { }</code>	<code>\$('p')</code>	Wählt alle Absätze im Dokument aus
ID	<code>#some-id { }</code>	<code>\$('#some-id')</code>	Wählt das einzelne Element im Dokument mit der ID <code>some-id</code> aus
Klasse	<code>.some-class { }</code>	<code>('.some-class { })</code>	Wählt alle Elemente im Dokument aus, denen die Klasse <code>some-class</code> zugeordnet ist

Wie in Kapitel 1, »Erste Schritte«, bereits erwähnt, werden beim Aufruf der Methoden eines jQuery-Objekts die Elemente, auf die an `$()` übergebenen Selektoren verweisen, automatisch und implizit in einer Schleife durchgegangen. Daher können wir die *explizite Iteration* gewöhnlich vermeiden, z.B. die for-Schleife, die in DOM-Skripte so häufig erforderlich ist.

Nachdem wir die Grundlagen besprochen haben, können wir uns jetzt einige nützlichere Verwendungen von Selektoren ansehen.

## 2.3 CSS-Selektoren

Die Bibliothek jQuery kann nahezu alle Selektoren aus den CSS-Spezifikationen 1 bis 3 verarbeiten, die auf der Website des World Wide Web Consortium unter <http://www.w3.org/Style/CSS/specs> aufgeführt werden. Dadurch können Entwickler ihre Websites verbessern, ohne sich Gedanken darüber machen zu müssen, welche Browser die komplexeren Selektoren nicht verstehen (vor allem Internet Explorer 6). In den Browsern muss nur JavaScript aktiviert sein.

### Fortschreitende Verbesserung

Verantwortungsbewusste jQuery-Entwickler sollten stets dem Prinzip der *fortschreitenden Verbesserung* bzw. der *allmählichen Funktionsverminderung* folgen, um dafür zu sorgen, dass die Seite bei ausgeschaltetem JavaScript zwar nicht so schön, aber genauso korrekt dargestellt wird wie mit aktiviertem JavaScript. Mit diesem Prinzip werden wir uns in diesem Buch durchgehend beschäftigen.

Um zu lernen, wie jQuery mit CSS-Selektoren funktioniert, verwenden wir eine Struktur, die auf vielen Websites eingesetzt wird, meistens für die Navigation – die geschachtelte, ungeordnete Liste:

```
<ul id="selected-plays">
  <li>Comedies
    <ul>
      <li><a href="/asyoulikeit/">As You Like It</a></li>
      <li>All's Well That Ends Well</li>
      <li>A Midsummer Night's Dream</li>
      <li>Twelfth Night</li>
    </ul>
  </li>
  <li>Tragedies
    <ul>
      <li><a href="hamlet.pdf">Hamlet</a></li>
      <li>Macbeth</li>
      <li>Romeo and Juliet</li>
    </ul>
  </li>
  <li>Histories
    <ul>
      <li>Henry IV (<a href="mailto:henryiv@king.co.uk">email</a>)</li>
        <ul>
          <li>Part I</li>
          <li>Part II</li>
        </ul>
      <li><a href="http://www.shakespeare.co.uk/henryv.htm">Henry V</a></li>
      <li>Richard II</li>
    </ul>
  </li>
</ul>
```

Beachten Sie, dass das erste <ul> die ID `selected-plays` hat, während keinem der <li>-Tags eine Klasse zugewiesen ist. Ohne Formatierung sieht die Liste etwa aus wie im folgenden Screenshot:



Die geschachtelte Liste erscheint so, wie wir es erwarten – als vertikal angeordnete Spiegelstrichaufzählung mit Einrückungen entsprechend der Aufzählungsebene.

### 2.3.1 Listenelemente formatieren

Nehmen wir an, dass wir die drei Elemente der obersten Ebene – und nur diese Elemente! – horizontal anordnen möchten. Dazu definieren wir als Erstes im Stylesheet die Klasse `horizontal`:

```
.horizontal {  
    float: left;  
    list-style: none;  
    margin: 10px;  
}
```

Die Klasse `horizontal` ordnet das Element »schwimmend« (als »Float-Element«) links neben dem nachfolgenden Element an, entfernt den Aufzählungspunkt von Listenelementen und fügt auf allen Seiten einen Rand von 10 Pixeln Breite hinzu.

Anstatt die Klasse `horizontal` direkt im HTML-Code zuzuweisen, fügen wir sie dynamisch nur den Listenelementen der obersten Ebene hinzu – `Comedies`, `Tragedies` und `Histories` –, um die Verwendung von Selektoren in jQuery zu zeigen:

```
$(document).ready(function() {  
    $('#selected-plays > li').addClass('horizontal');  
});
```

**Listing 2-1**

Wie in Kapitel 1 besprochen, beginnen wir den jQuery-Code, indem wir `$(document).ready()` aufrufen, wodurch die übergebene Funktion ausgeführt wird, sobald das DOM geladen ist, aber nicht früher.

In der zweiten Zeile wird der *Kindkombinator* (`>`) verwendet, um die Klasse `horizontal` ausschließlich den Elementen der obersten Ebene zuzuweisen. Der Selektor innerhalb der Funktion `$()` besagt letzten Endes: »Finde jedes Listenelement (`li`), das ein Kind (`>`) des Elements mit der ID `selected-plays` ist (`#selected-plays`).«

Nachdem die Klasse zugewiesen ist, treten die Regeln in Kraft, die dafür im Stylesheet definiert sind. Unsere geschachtelte Liste sieht jetzt wie in der folgenden Abbildung aus:

Comedies	Tragedies	Histories
<ul style="list-style-type: none"><li>○ <a href="#">As You Like It</a></li><li>○ All's Well That Ends Well</li><li>○ A Midsummer Night's Dream</li><li>○ Twelfth Night</li></ul>	<ul style="list-style-type: none"><li>○ <a href="#">Hamlet</a></li><li>○ Macbeth</li><li>○ Romeo and Juliet</li></ul>	<ul style="list-style-type: none"><li>○ Henry IV (email)<ul style="list-style-type: none"><li>■ Part I</li><li>■ Part II</li></ul></li><li>○ <a href="#">Henry V</a></li><li>○ Richard II</li></ul>

Die Formatierung aller anderen Elemente – also derjenigen, die sich nicht auf der obersten Ebene befinden – kann auf verschiedene Weise erfolgen. Da wir bereits die Klasse `horizontal` auf die Elemente der obersten Ebene angewandt haben, besteht eine Möglichkeit zur Auswahl aller Elemente der darunter liegenden Ebenen darin, eine *Negations-Pseudoklasse* zur Bezeichnung aller Listenelemente zu verwenden, die nicht die Klasse `horizontal` haben. Beachten Sie die neu hinzugefügte dritte Codezeile:

```
$(document).ready(function() {
    $('#selected-plays > li').addClass('horizontal');
    $('#selected-plays li:not(.horizontal)').addClass('sub-level');
});
```

#### ***Listing 2-2***

Diesmal wählen wir jedes Listenelement (`<li>`) aus, das folgende Eigenschaften aufweist:

- Es ist ein Nachfahr des Elements mit der ID `selected-plays` (`#selected-plays`)
- Es hat nicht die Klasse `horizontal` (`:not(.horizontal)`)

Wenn wir diesen Elementen die Klasse `sub-level` zuweisen, erhalten sie den im Stylesheet definierten Hintergrund. Jetzt sieht unsere geschachtelte Liste wie folgt aus:

### Selected Shakespeare Plays

<p>Comedies</p> <ul style="list-style-type: none"><li>o <a href="#">As You Like It</a></li><li>o All's Well That Ends Well</li><li>o A Midsummer Night's Dream</li><li>o Twelfth Night</li></ul>	<p>Tragedies</p> <ul style="list-style-type: none"><li>o Hamlet</li><li>o Macbeth</li><li>o Romeo and Juliet</li></ul>	<p>Histories</p> <ul style="list-style-type: none"><li>o Henry IV (<a href="#">email</a>)<ul style="list-style-type: none"><li>■ Part I</li><li>■ Part II</li></ul></li><li>o Henry V</li><li>o Richard II</li></ul>
--	--	--

### 2.3.2 Attributselektoren

*Attributselektoren* sind eine besonders praktische Art von CSS-Selektoren. Mit ihrer Hilfe können wir ein Element anhand eines seiner HTML-Attribute bezeichnen, z.B. mit dem Attribut `title` bei einem Link oder mit dem Attribut `alt` bei einem Bild. Um zum Beispiel alle Bilder auszuwählen, die über das Attribut `alt` verfügen, schreiben wir Folgendes:

```
$( 'img[alt]' )
```

Attributselektoren akzeptieren auch eine Syntax von Jokerzeichen, die regulären Ausdrücken entlehnt ist, um einen Wert am Anfang (^) oder Ende (\$) eines Strings zu bezeichnen. Weitere mögliche Zeichen sind das Sternchen (\*) für einen Wert an einer beliebigen Position in einem String sowie das Ausrufezeichen (!) für einen negierten Wert.

### 2.3.3 Links formatieren

Nehmen wir an, dass wir für die verschiedenen Arten von Links jeweils unterschiedliche Formate haben möchten. Als Erstes definieren wir dazu diese Formate in unserem Stylesheet:

```
a {  
    color: #00c;  
}  
a.mailto {  
    background: url(images/mail.png) no-repeat right top;  
    padding-right: 18px;  
}  
a.pdflink {  
    background: url(images/pdf.png) no-repeat right top;  
    padding-right: 18px;  
}  
a.henrylink {  
    background-color: #fff;  
    padding: 2px;  
    border: 1px solid #000;  
}
```

Anschließend fügen wir die drei Klassen – mailto, pdflink und henrylink – mithilfe von jQuery den entsprechenden Links hinzu.

Um eine Klasse für alle E-Mail-Links hinzuzufügen, konstruieren wir einen Selektor, der nach allen Ankerelementen (a) sucht, deren href-Attribut ([href]) mit mailto: beginnt (^="mailto:"):

```
$(document).ready(function() {  
    $('a[href^="mailto:"]').addClass('mailto');  
});
```

**Listing 2–3**

Aufgrund der im Stylesheet der Seite definierten Regeln erscheint hinter allen mailto-Links auf der Seite ein Briefumschlag, wie die folgende Abbildung zeigt:

The screenshot shows a list of Shakespeare plays categorized into Comedies, Tragedies, and Histories. The 'mailto:' links in the Comedies and Histories sections are styled with a small envelope icon.

Selected Shakespeare Plays		
Comedies	Tragedies	Histories
o <a href="#">As You Like It</a> o <a href="#">All's Well That Ends Well</a> o <a href="#">A Midsummer Night's Dream</a> o <a href="#">Twelfth Night</a>	o <a href="#">Hamlet</a> o <a href="#">Macbeth</a> o <a href="#">Romeo and Juliet</a>	o <a href="#">Henry IV</a> (email  ) ■ Part I ■ Part II o <a href="#">Henry V</a> o <a href="#">Richard II</a>

Um eine Klasse für alle Links zu PDF-Dateien hinzuzufügen, verwenden wir nicht den Zirkumflex, sondern das Dollarzeichen, da wir nach Links suchen, deren href-Attribut mit .pdf endet:

```
$(document).ready(function() {  
    $('a[href^="mailto:"]').addClass('mailto');  
    $('a[href$=".pdf"]').addClass('pdflink');  
});
```

**Listing 2–4**

Die Stylesheet-Regel für die neu hinzugefügte Klasse pdflink führt dazu, dass hinter jedem Link zu einem PDF-Dokument ein Adobe Acrobat-Symbol erscheint:

The screenshot shows the same list of Shakespeare plays as before, but with the addition of the 'pdflink' class. The PDF links now have a small Adobe Acrobat icon.

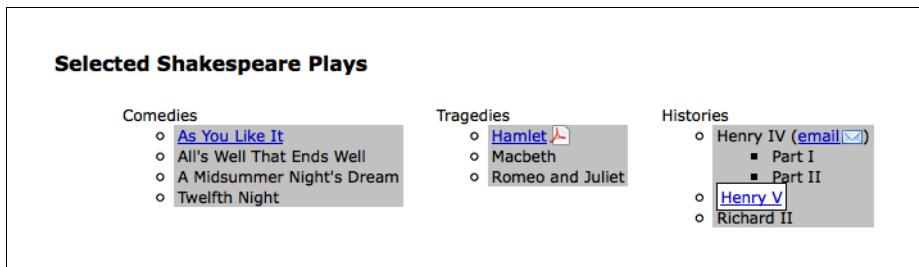
Selected Shakespeare Plays		
Comedies	Tragedies	Histories
o <a href="#">As You Like It</a> o <a href="#">All's Well That Ends Well</a> o <a href="#">A Midsummer Night's Dream</a> o <a href="#">Twelfth Night</a>	o <a href="#">Hamlet</a>  o <a href="#">Macbeth</a> o <a href="#">Romeo and Juliet</a>	o <a href="#">Henry IV</a> (email  ) ■ Part I ■ Part II o <a href="#">Henry V</a> o <a href="#">Richard II</a>

Attributselektoren lassen sich auch kombinieren. Beispielsweise können wir die Klasse `henrylink` zu allen Links hinzufügen, deren `href`-Wert mit `http` beginnt und an irgendeiner Stelle die Zeichenfolge `henry` enthält:

```
$(document).ready(function() {  
    $('a[href^="mailto:"]').addClass('mailto');  
    $('a[href$=".pdf"]').addClass('pdflink');  
    $('a[href^="http"] [href*="henry"]')  
        .addClass('henrylink');  
});
```

**Listing 2–5**

Nachdem alle drei Klassen zu den drei Arten von Links hinzugefügt worden sind, sieht unsere Seite folgendermaßen aus:



Beachten Sie in dem vorstehenden Screenshot das PDF-Symbol rechts neben dem *Hamlet*-Link, das Briefumschlagsymbol neben dem E-Mail-Link und den weißen Hintergrund und den schwarzen Rahmen um den Link *Henry V*.

## 2.4 jQuery-Selektoren

Zu der breiten Palette von CSS-Selektoren fügt jQuery noch seine eigenen hinzu, die die bereits beeindruckenden Möglichkeiten der CSS-Selektoren erweitern, um Elemente auf der Seite auf ganz neuen Wegen zu finden.

### Ein Hinweis zur Performance

Wenn möglich, verwendet jQuery das browsereigene DOM-Selektormodul, um Elemente zu finden, doch diese sehr schnelle Vorgehensweise kann nicht verwendet werden, wenn jQuery-Selektoren zum Einsatz kommen. Aus diesem Grund sollten Sie jQuery-Selektoren vermeiden, wenn es eine systemeigene Alternative gibt und die Performance eine Rolle spielt.

Die meisten jQuery-Selektoren ermöglichen es uns, bestimmte Elemente sozusagen aus einer Aneinanderreihung auszuwählen. Gewöhnlich werden sie nach einem CSS-Selektor angewendet, um Elemente anhand ihrer Position in der zuvor ausgewählten Gruppe zu bestimmen. Die Syntax ist identisch mit der *Pseudoklassen-Syntax* von CSS, bei der der Selektor mit einem Doppelpunkt (:) beginnt. Um beispielsweise das zweite Element aus einem Satz von <div>-Elementen mit der Klasse horizontal auszuwählen, schreiben wir folgenden Code:

```
$('.div.horizontal:eq(1)')
```

Beachten Sie, dass :eq(1) das zweite Element in diesem Satz auswählt, da die Nummerierung in JavaScript-Arrays bei null beginnt. Im Gegensatz dazu fängt die Nummerierung bei CSS mit 1 an, weshalb ein CSS-Selektor wie \$('div:nth-child(1)') alle div-Selektoren auswählt, die das erste Kind ihres Elternelements sind. (In einem solchen Fall würden wir allerdings eher '\$div:first-child' verwenden.)

#### 2.4.1 Zeilen abwechselnd formatieren

Zwei sehr nützliche jQuery-Selektoren sind :odd und :even. Sehen wir uns nun an, wie wir sie verwenden können, um die folgenden Tabellen mit einem einfachen »Streifenmuster« zu versehen:

```
<h2>Shakespeare's Plays</h2>
<table>
  <tr>
    <td>As You Like It</td>
    <td>Comedy</td>
    <td></td>
  </tr>
  <tr>
    <td>All's Well that Ends Well</td>
    <td>Comedy</td>
    <td>1601</td>
  </tr>
  <tr>
    <td>Hamlet</td>
    <td>Tragedy</td>
    <td>1604</td>
  </tr>
  <tr>
    <td>Macbeth</td>
    <td>Tragedy</td>
    <td>1606</td>
  </tr>
```

```
<tr>
    <td>Romeo and Juliet</td>
    <td>Tragedy</td>
    <td>1595</td>
</tr>
<tr>
    <td>Henry IV, Part I</td>
    <td>History</td>
    <td>1596</td>
</tr>
<tr>
    <td>Henry V</td>
    <td>History</td>
    <td>1599</td>
</tr>
</table>
<h2>Shakespeare's Sonnets</h2>
<table>
<tr>
    <td>The Fair Youth</td>
    <td>1–126</td>
</tr>
<tr>
    <td>The Dark Lady</td>
    <td>127–152</td>
</tr>
<tr>
    <td>The Rival Poet</td>
    <td>78–86</td>
</tr>
</table>
```

Mit der minimalen Formatierung in unserem Stylesheet erscheinen die Überschriften und Tabellen sehr schmucklos. Die Tabelle hat einen weißen Hintergrund, doch gibt es kein Format, das eine Zeile von der nächsten absetzt:

Shakespeare's Plays	
As You Like It	Comedy
All's Well that Ends Well	Comedy 1601
Hamlet	Tragedy 1604
Macbeth	Tragedy 1606
Romeo and Juliet	Tragedy 1595
Henry IV, Part I	History 1596
Henry V	History 1599
Shakespeare's Sonnets	
The Fair Youth	1–126
The Dark Lady	127–152
The Rival Poet	78–86

Wir können dem Stylesheet jetzt ein Format für alle Tabellenzeilen hinzufügen und die Klasse `alt` für die ungeradzahligen Zeilen verwenden:

```
tr {  
    background-color: #fff;  
}  
.alt {  
    background-color: #ccc;  
}
```

Schließlich schreiben wir unseren jQuery-Code, indem wir die Klasse zu den Tabellenzeilen (`<tr>`-Tags) mit ungerader Nummer hinzufügen:

```
$(document).ready(function() {  
    $('tr:even').addClass('alt');  
});
```

**Listing 2–6**

Moment mal! Wieso verwenden wir hier den Selektor `:even` für die ungeradzahligen Zeilen? Nun, genau wie `:eq()` verwenden auch die Selektoren `:even` und `:odd` die bei null beginnende JavaScript-Nummerierung. Daher wird die erste Zeile als Zeile 0 gezählt (gerade), die zweite als Zeile 1 (ungerade) usw. Unser einfacher Code führt daher zu dem folgenden Erscheinungsbild unserer Tabellen:

Shakespeare's Plays		
As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets		
The Fair Youth	1–126	
The Dark Lady	127–152	
The Rival Poet	78–86	

Das Ergebnis für die zweite Tabelle ist nicht unbedingt das, was wir uns vorgestellt haben. Da die letzte Zeile der Tabelle mit den Schauspielern den »alternativen« grauen Hintergrund hat, wird für die erste Zeile der Sonettabelle der einfache weiße Hintergrund verwendet. Eine Möglichkeit, um dieses Problem zu verhindern, besteht darin, den Selektor `:nth-child()` zu verwenden, der die Elementposition relativ zum Elternelement zählt und nicht relativ zu allen bisher ausgewählten Objekten. Dieser Selektor kann eine Zahl, aber auch die Angaben `odd` und `even` als Argument übernehmen.

```
$(document).ready(function() {  
    $('tr:nth-child(odd)').addClass('alt');  
});
```

**Listing 2–7**

Beachten Sie, dass `:nth-child()` der einzige jQuery-Selektor ist, der beim Zählen bei eins beginnt. Um dasselbe Streifenmuster zu erhalten wie vorher – mit Ausnahme der Weiterführung in der zweiten Tabelle –, müssen wir als Argument hier `odd` verwenden statt `even`. Mit diesem Selektor werden beide Tabellen jetzt mit einem passenden Streifenmuster dargestellt, wie die folgende Abbildung zeigt:

Shakespeare's Plays		
As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
Henry IV, Part I	History	1596
Henry V	History	1599

Shakespeare's Sonnets		
The Fair Youth	1-126	
The Dark Lady	127-152	
The Rival Poet	78-86	

Um dem Ganzen noch mithilfe von jQuery-Selektoren den letzten Schliff zu geben, nehmen wir an, dass wir aus irgendeinem Grunde alle Tabellenzellen hervorheben möchten, in denen es um eines der *Henry*-Stücke geht. Dazu müssen wir natürlich eine Klasse zum Stylesheet hinzufügen, um den Text fett und kursiv darzustellen (`.highlight {font-weight: bold; font-style: italic;}`), ansonsten aber nicht mehr tun, als unserem jQuery-Code eine einzige Zeile mit dem Selektor `:contains()` hinzuzufügen, wie der folgende Codeausschnitt zeigt:

```
$(document).ready(function() {
    $('tr:nth-child(odd)').addClass('alt');
    $('td:contains(Henry)').addClass('highlight');
});
```

**Listing 2-8**

Jetzt sehen wir unsere schön gestreifte Tabelle mit den hervorgehobenen *Henry*-Stücken:

Shakespeare's Plays		
As You Like It	Comedy	
All's Well that Ends Well	Comedy	1601
Hamlet	Tragedy	1604
Macbeth	Tragedy	1606
Romeo and Juliet	Tragedy	1595
<b><i>Henry IV, Part I</i></b>	History	1596
<b><i>Henry V</i></b>	History	1599

Shakespeare's Sonnets		
The Fair Youth	1-126	
The Dark Lady	127-152	
The Rival Poet	78-86	

Sehr wichtig ist der Umstand, dass der Selektor `:contains()` zwischen Groß- und Kleinschreibung unterscheidet. Hätten wir `$('td:contains(henry)')` mit kleinem `h` geschrieben, wären keine Zellen ausgewählt worden.

Zugegeben, es gibt Möglichkeiten, die Streifendarstellung von Tabellen und die Hervorhebung von Text auch ohne jQuery oder irgendeine Form von clientseitiger Programmierung zu erzielen. Trotzdem ist jQuery in Kombination mit CSS eine großartige Alternative für diese Art der Formatierung, wenn der Inhalt dynamisch generiert wird oder wenn wir keinen Zugriff auf den HTML- oder den serverseitigen Code haben.

### 2.4.2 Formularselektoren

jQuery-Selektoren sind nicht darauf beschränkt, Elemente auf der Grundlage ihrer Position zu finden. Wenn Sie mit Formularen arbeiten, können Ihnen die Selektoren von jQuery und die ergänzenden CSS3-Selektoren beispielsweise dabei helfen, ausschließlich die Elemente auszuwählen, die Sie brauchen. Die folgende Tabelle beschreibt eine Hand voll dieser *Formularselektoren*:

Selektor	Gefundene Elemente
<code>:input</code>	Eingabefelder, Textbereiche, Auswahl- und Schaltflächenelemente
<code>:button</code>	Schaltflächen- und Eingabeelemente, deren <code>type</code> -Attribut den Wert <code>button</code> hat
<code>:enabled</code>	Aktivierte Formularelemente
<code>:disabled</code>	Deaktivierte Formularelemente
<code>:checked</code>	Aktivierte Optionsschalter und Kontrollkästchen
<code>:selected</code>	Ausgewählte Optionselemente

Wie andere Selektoren können auch Formularselektoren kombiniert werden, um die Suche spezifischer zu gestalten. Beispielsweise können wir mit `$('input[type="radio"] :checked')` nach allen aktivierten Optionsschaltern (ohne aktivierte Kontrollkästchen!) suchen oder mit `$('input[type="password"]', 'input[type="text"]') :disabled'` nach allen Kennwort-Eingabeelementen und deaktivierten Texteingabeelementen. Auch bei jQuery-Selektoren verwenden wir die CSS-Grundprinzipien, um zu bestimmen, welche Elemente gefunden werden.

Wir haben bis jetzt nur an der Oberfläche der verfügbaren Selektorausdrücke gekratzt. In Kapitel 9, »Komplexe Selektoren und Durchlaufen des DOM«, beschäftigen wir uns eingehender mit diesem Thema.

## 2.5 Methoden zum Durchlaufen des DOM

Die jQuery-Selektoren, die wir uns bis jetzt angesehen haben, erlauben uns, einen Satz von Elementen auszuwählen, während wir uns im DOM-Baum seitwärts und abwärts bewegen und die Ergebnisse filtern. Wenn das der einzige Weg wäre, um Elemente auszuwählen, wären unsere Möglichkeiten ziemlich begrenzt (obwohl die Selektorausdrücke an sich schon umfassend sind, vor allem im Vergleich mit regulären Optionen für die DOM-Skripterstellung). Es gibt viele Situationen, in denen es von entscheidender Bedeutung ist, ein Eltern- oder Vorfahren-element auszuwählen. Hier kommen die jQuery-Methoden zum Durchlaufen des DOM ins Spiel. Damit können wir uns mit Leichtigkeit aufwärts und abwärts und durch den gesamten DOM-Baum bewegen.

Einige der Methoden verfügen über fast identische Gegenstücke unter den Selektorausdrücken. Beispielsweise könnten wir die Zeile `$('tr:even').addClass('alt')`, die wir als Erstes verwendet haben, um die Klasse alt hinzuzufügen, wie folgt mit der Methode `.filter()` umschreiben:

```
$('tr').filter(':even').addClass('alt');
```

Meistens ergänzen die beiden Alternativen zur Elementauswahl einander. Außerdem bietet gerade die Methode `.filter()` enorme Möglichkeiten, da sie als Argument eine Funktion annehmen kann, mit der wir komplexe Tests zusammenstellen können, um zu bestimmen, welche Elemente in der Menge der Übereinstimmungen verbleiben sollen. Nehmen wir beispielsweise an, dass wir allen externen Links eine Klasse hinzufügen möchten. Dafür gibt es in jQuery keinen Selektor. Ohne die Filterfunktion wären wir gezwungen, jedes Element in einer Schleife durchzugehen und einzeln zu testen. Mit der folgenden *Filterfunktion* dagegen können wir uns auf die implizite Iteration von jQuery stützen und unseren Code kompakt halten:

```
$('a').filter(function() {
    return this.hostname && this.hostname != location.hostname;
}).addClass('external');
```

### **Listing 2-9**

Die zweite Zeile filtert die Menge der `<a>`-Elemente nach zwei Kriterien:

1. Sie müssen über ein `href`-Attribut mit einem Domänennamen verfügen (`this.hostname`). Mit diesem Test schließen wir beispielsweise `mailto`-Links aus.
2. Der Domänenname, zu dem der Link führt (wiederum `this.hostname`) darf nicht (`!=`) mit dem Domänennamen der aktuellen Seite (`location.hostname`) übereinstimmen.

Genauer gesagt, die Methode `.filter()` iteriert durch die Menge übereinstimmender Elemente, ruft für jedes davon die Funktion auf und prüft den Rückgabewert. Wenn die Funktion `false` zurückgibt, wird das Element aus der Menge der Übereinstimmungen entfernt. Lautet er `true`, wird das Element beibehalten:

Selected Shakespeare Plays		
Comedies <ul style="list-style-type: none"><li>o <a href="#">As You Like It</a></li><li>o All's Well That Ends Well</li><li>o A Midsummer Night's Dream</li><li>o Twelfth Night</li></ul>	Tragedies <ul style="list-style-type: none"><li>o <a href="#">Hamlet ↗</a></li><li>o Macbeth</li><li>o Romeo and Juliet</li></ul>	Histories <ul style="list-style-type: none"><li>o <a href="#">Henry IV (email ↗)</a><ul style="list-style-type: none"><li>▪ Part I</li><li>▪ Part II</li></ul></li><li>o <a href="#">Henry V ↗</a></li><li>o Richard II</li></ul>

Im nächsten Abschnitt kommen wir wieder auf unsere gestreiften Tabellen zurück, um uns anzusehen, was mit den Traversierungsmethoden noch möglich ist.

### 2.5.1 Einzelne Zellen formatieren

Zuvor haben wir bereits die Klasse `highlight` zu allen Zellen hinzugefügt, die den Text `Henry` enthalten. Um die Zelle daneben zu formatieren, können wir den bereits geschriebenen Selektor verwenden und einfach die Methode `.next()` für das Ergebnis aufrufen:

```
$(document).ready(function() {  
    $('td:contains(Henry)').next().addClass('highlight');  
});
```

**Listing 2-10**

Die Tabelle sieht jetzt wie in der folgenden Abbildung aus:

Shakespeare's Plays			
As You Like It	Comedy		
All's Well that Ends Well	Comedy	1601	
Hamlet	Tragedy	1604	
Macbeth	Tragedy	1606	
Romeo and Juliet	Tragedy	1595	
Henry IV, Part I	<b>History</b>	1596	
Henry V	<b>History</b>	1599	
Shakespeare's Sonnets			
The Fair Youth	1-126		
The Dark Lady	127-152		
The Rival Poet	78-86		

Die Methode `.next()` wählt nur das nächstliegende Geschwisterelement aus. Um alle Zellen hervorzuheben, die auf eine Zelle mit dem Inhalt `Henry` folgen, verwenden wir stattdessen `.nextAll()`:

```
$(document).ready(function() {
    $('td:contains(Henry)').nextAll().addClass('highlight');
});
```

**Listing 2-11**

Da sich die Zellen mit dem Text `Henry` in der ersten Tabellenspalte befinden, führt dieser Code dazu, dass alle folgenden Zellen in der betreffenden Zeile hervorgehoben werden, wie die Abbildung zeigt:

Shakespeare's Plays	
As You Like It	Comedy
All's Well that Ends Well	Comedy 1601
Hamlet	Tragedy 1604
Macbeth	Tragedy 1606
Romeo and Juliet	Tragedy 1595
Henry IV, Part I	<b>History 1596</b>
Henry V	<b>History 1599</b>
Shakespeare's Sonnets	
The Fair Youth 1–126	
The Dark Lady 127–152	
The Rival Poet 78–86	

Wie zu erwarten ist, gibt es zu den Methoden `.next()` und `.nextAll()` die Gegenstücke `.prev()` und `.prevAll()`. Außerdem gibt es die Methode `.siblings()`, die alle Elemente auf derselben DOM-Ebene unabhängig davon auswählt, ob sie dem zuvor ausgewählten Element vorausgehen oder ihm nachfolgen.

Um auch die ursprüngliche Zelle (die mit dem Text `Henry`) zusammen mit den nachfolgenden auszuwählen, fügen wir wie folgt die Methode `.andSelf()` hinzu:

```
$(document).ready(function() {
    $('td:contains(Henry)').nextAll().andSelf()
        .addClass('highlight');
});
```

**Listing 2-12**

Durch diese Änderung werden jetzt alle Zellen in der Zeile mit dem Format der Klasse `highlight` versehen:

Shakespeare's Plays	
As You Like It	Comedy
All's Well that Ends Well	Comedy 1601
Hamlet	Tragedy 1604
Macbeth	Tragedy 1606
Romeo and Juliet	Tragedy 1595
<b>Henry IV, Part I</b>	<b>History 1596</b>
<b>Henry V</b>	<b>History 1599</b>

Shakespeare's Sonnets	
The Fair Youth 1–126	
The Dark Lady 127–152	
The Rival Poet 78–86	

Um eines klarzustellen: Es gibt viele Kombinationen von Selektoren und Traversierungsmethoden, um dieselbe Menge von Elementen abzurufen. Der folgende Code zeigt beispielsweise eine andere Möglichkeit, um jede Zeile auszuwählen, in der mindestens eine der Zellen den Text `Henry` enthält:

```
$(document).ready(function() {
    $('td:contains(Henry)').parent().children()
        .addClass('highlight');
});
```

#### ***Listing 2-13***

Wir durchlaufen jetzt nicht die Geschwisterelemente, sondern gehen mit `.parent()` im DOM eine Ebene höher zum Element `<tr>` und wählen mit `.children()` alle Zellen dieser Zeile aus.

### **2.5.2 Verkettung**

Die Kombinationen der Traversierungsmethoden, die wir uns gerade angesehen haben, veranschaulichen die Möglichkeiten von jQuery zur *Verkettung*. jQuery erlaubt es, in einer einzigen Codezeile mehrere Sätze von Elementen auszuwählen und mehrere verschiedene Dinge mit ihnen anzustellen. Diese Verkettung hilft nicht nur dabei, den jQuery-Code knapp zu halten, sondern kann auch die Performance eines Skripts verbessern, wenn dadurch vermieden wird, einen Selektor erneut zu spezifizieren.

#### **Wie die Verkettung funktioniert**

Wie die Verkettung implementiert wird, sehen wir uns noch ausführlich an. Vorläufig stellen Sie sich einfach vor, dass fast alle jQuery-Methoden ein jQuery-Objekt zurückgeben, sodass auf das Ergebnis weitere jQuery-Methoden angewandt werden können.

Es ist auch möglich, eine einzelne Codezeile in mehrere aufzuteilen, um die Lesbarkeit zu erhöhen. Eine verkettete Abfolge von Methoden können wir wie folgt in einer Zeile schreiben:

```
$('td:contains(Henry)').parent().find('td:eq(1)')  
    .addClass('highlight').end().find('td:eq(2)')  
    .addClass('highlight');
```

**Listing 2-14**

Es ist aber auch möglich, daraus sieben Zeilen zu machen:

```
$('td:contains(Henry)') // Findet jede Zelle mit "Henry"  
    .parent() // Wählt das Elternelement aus  
    .find('td:eq(1)') // Findet die 2. Nachfahrendelle  
    .addClass('highlight') // Fügt die Klasse highlight hinzu  
    .end() // Kehrt zum Elternelement der Zelle mit Henry zurück  
    .find('td:eq(2)') // Findet die 3. Nachfahrendelle  
    .addClass('highlight'); // Fügt die Klasse highlight hinzu
```

**Listing 2-15**

Zugegeben, der Durchlauf durch das DOM ist in diesem Beispiel absurd umständlich. So etwas sollten Sie nicht verwenden, da deutlich einfachere und unmittelbarere Methoden zur Verfügung stehen. Dieses Beispiel soll lediglich zeigen, was für eine Flexibilität uns die Verkettung bietet.

Die Verkettung wirkt so ähnlich, wie einen ganzen Absatz in einem Atemzug zu lesen – die Aufgabe wird schnell erledigt, ist aber für Außenstehende schwer verständlich. Die Zerlegung in mehrere Zeilen und die Ergänzung um sinnvolle Kommentare kann langfristig Zeit sparen.

## 2.6 Zugriff auf DOM-Elemente

Alle Selektorausdrücke und die meisten jQuery-Methoden geben ein jQuery-Objekt zurück. Das ist fast immer genau das, was wir wollen, da ein solches Objekt eine implizite Iteration und die Möglichkeit der Verkettung bietet.

Es gibt jedoch auch Stellen in unserem Code, in denen wir direkt auf ein *DOM-Element* zugreifen müssen, beispielsweise wenn wir eine Ergebnismenge von Elementen für eine andere JavaScript-Bibliothek zur Verfügung stellen müssen. Außerdem kann es sein, dass wir auf den Tag-Namen eines Elements zugreifen müssen, der als *Eigenschaft* des DOM-Elements vorliegt. Für diese zugegebenermaßen seltenen Situationen bietet jQuery die Methode `.get()`. Um auf das erste DOM-Element zuzugreifen, das das jQuery-Objekt verwendet, benutzen wir `.get(0)`. Befindet sich das benötigte DOM-Element in einer Schleife, nehmen wir `.get(index)`. Um also den Tag-Namen des Elements mit `id="my-element"` herauszufinden, schreiben wir folgenden Code:

```
var myTag = $('#my-element').get(0).tagName;
```

Noch einfacher wird es dadurch, dass jQuery eine Abkürzung für `.get()` bietet. Statt der vorstehenden Zeilen können wir eckige Klammern verwenden, auf die unmittelbar der Selektor folgt:

```
var myTag = $('#my-element')[0].tagName;
```

Es ist kein Zufall, dass diese Syntax so aussieht, als würde das jQuery-Objekt wie ein Array aus DOM-Elementen behandelt. Durch die eckigen Klammern wird förmlich der jQuery-Wrapper abgeschält, um an die Knotenliste zu kommen, und durch die Einbeziehung des Index (in diesem Fall 0) wird praktisch das DOM-Element selbst herausgefischt.

## 2.7 Zusammenfassung

Mit den in diesem Kapitel behandelten Techniken sind wir in der Lage, Mengen von Elementen auf verschiedene Weise auf der Seite zu finden. Vor allem haben wir gelernt, wie wir Elemente der obersten und der folgenden Ebene in einer geschachtelten Liste mithilfe grundlegender CSS-Selektoren formatieren, wie wir unterschiedliche Arten von Links mithilfe von Attributselektoren verschieden gestalten, wie wir mit den jQuery-Selektoren `:odd` und `:even` oder mit dem ausgefilterten CSS-Selektor `:nth-child()` eine einfache Streifengestaltung für Tabellen erreichen, und wie wir Text in bestimmten Tabellenzellen durch die Verkettung von jQuery-Methoden hervorheben.

Bis jetzt haben wir die Methode `$(document).ready()` verwendet, um eine Klasse zu einer Ergebnismenge von Elementen hinzuzufügen. Im nächsten Kapitel sehen wir uns andere Möglichkeiten an, um dies als Reaktion auf verschiedene vom Benutzer ausgelöste Ereignisse zu tun.

### 2.7.1 Literatur

Das Thema Selektoren und Traversierungsmethoden wird ausführlicher in Kapitel 9 behandelt. Eine vollständige Liste der Selektoren und Traversierungsmethoden von jQuery finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation auf <http://api.jquery.com/>.

## 2.8 Übungsaufgaben

Um die folgenden Übungen durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Fügen Sie die Klasse `special` zu allen `<li>`-Elementen auf der zweiten Ebene der geschachtelten Liste hinzu.
2. Fügen Sie die Klasse `year` zu allen Zellen in der dritten Tabellenspalte hinzu.
3. Fügen Sie die Klasse `special` zur ersten Tabellenzelle hinzu, in der das Wort `tragedy` vorkommt.
4. Schwierig: Wählen Sie die Listenelemente (`<li>`) aus, die einen Link (`<a>`) enthalten. Fügen Sie die Klasse `afterlink` zu den Geschwisterlinkelementen hinzu, die auf die ausgewählten folgen.
5. Schwierig: Fügen Sie die Klasse `tragedy` jeweils zum nächstliegenden `<ul>`-Vorfahren aller `.pdf`-Links hinzu.



## 3 Ereignisbehandlung

JavaScript hat mehrere eingebaute Möglichkeiten, um auf Benutzeraktionen und andere Ereignisse zu reagieren. Um eine Seite dynamisch und reaktiv zu gestalten, müssen wir diese Möglichkeiten nutzen, damit wir die bis jetzt gelernten jQuery-Techniken und die anderen Tricks, die wir noch behandeln werden, jeweils zur richtigen Zeit anwenden können. Das wäre zwar auch mit 08/15-JavaScript möglich, doch jQuery verbessert und erweitert die grundlegenden Mechanismen zur Ereignisbehandlung, um ihnen eine gefälligere Syntax zu geben und sie gleichzeitig leistungsfähiger zu machen.

### 3.1 Aufgaben beim Laden der Seite durchführen

Wir haben bereits erfahren, wie wir jQuery dazu bringen, auf das Laden einer Webseite zu reagieren. Der Ereignishandler `$(document).ready()` kann verwendet werden, um den Code für eine ganze Funktion auszulösen, aber er kann noch mehr.

#### 3.1.1 Die Codeausführung zeitlich abstimmen

In Kapitel 1, »Erste Schritte«, haben wir festgestellt, dass `$(document).ready()` die wichtigste Möglichkeit von jQuery ist, um Aufgaben beim Laden der Seite durchzuführen. Es ist jedoch nicht die einzige Methode, die uns zur Verfügung steht. Mit dem nativen Ereignis `window.onload` können wir ein ähnliches Ergebnis erzielen. Die beiden Methoden sind sich zwar ziemlich ähnlich, doch ist es wichtig, sich über die Unterschiede im Timing im Klaren zu sein, auch wenn sie je nach der Anzahl der zu ladenden Ressourcen sehr gering sein können.

Das Ereignis `window.onload` wird ausgelöst, wenn ein Dokument komplett in den Browser heruntergeladen ist. Das bedeutet, dass jedes Element auf der Seite für die Bearbeitung durch JavaScript bereitsteht. Das ist wichtig, um Code mit vielen Funktionen zu schreiben, ohne sich Sorgen um die Reihenfolge des Ladens zu machen.

Dagegen wird ein mit `$(document).ready()` registrierter Handler aufgerufen, wenn das DOM vollständig verwendungsbereit ist. Das bedeutet, dass alle Elemente für unsere Skripte zugänglich sind, aber nicht, dass jede zugehörige Datei geladen sein muss. Sobald der HTML-Code heruntergeladen und zu einem DOM-Baum geparsst wurde, kann der Code ausgeführt werden.

### Laden der Formatierungen und Codeausführung

Um sicherzustellen, dass die Seite formatiert wird, bevor der JavaScript-Code läuft, sollten Sie die `<link rel="stylesheet">`- und `<style>`-Tags innerhalb des `<head>`-Elements für das Dokument vor den `<script>`-Tags platzieren.

Nehmen wir als Beispiel eine Seite mit einer Bildergalerie. Darauf können viele große Bilder vorhanden sein, die wir mit jQuery ausblenden, anzeigen, verschieben oder auf andere Weise manipulieren können. Wenn wir unsere Schnittstelle so einrichten, dass sie das Ereignis `onload` verwendet, müssen die Benutzer warten, bis alle Bilder komplett heruntergeladen sind, bevor sie die Funktionen nutzen können. Was noch schlimmer ist: Wenn Verhaltensweisen noch nicht Elementen zugewiesen sind, die über ein Standardverhalten verfügen (z.B. Links), können Benutzeraktionen zu unvorhergesehenen Ergebnissen führen. Verwenden wir dagegen `$(document).ready()` für die Einrichtung, steht die Schnittstelle schon viel eher und zudem mit dem richtigen Verhalten zur Verfügung.

### Was ist geladen und was nicht?

Die Verwendung von `$(document).ready()` ist fast immer gegenüber einem `onload`-Handler zu bevorzugen. Allerdings müssen wir daran denken, dass die unterstützten Dateien möglicherweise noch nicht geladen sind, weshalb Attribute wie Bildhöhen und -breiten zu diesem Zeitpunkt nicht unbedingt zur Verfügung stehen. Wenn diese Parameter gebraucht werden, können wir gelegentlich auch einen `onload`-Handler implementieren (oder besser jQuery verwenden, um einen Handler an ein `load`-Ereignis zu binden). Die beiden Mechanismen können friedlich nebeneinander bestehen.

#### 3.1.2 Mehrere Skripte auf einer Seite

Der herkömmliche Mechanismus zur Registrierung von Ereignishandlern mithilfe von JavaScript (anstatt gleich im HTML-Code Handlerattribute hinzuzufügen) besteht darin, eine Funktion dem entsprechenden Attribut des DOM-Elements zuzuweisen. Nehmen wir an, wir haben folgende Funktion definiert:

```
function doStuff() {  
    // Führt eine Aufgabe durch ...  
}
```

Die Zuweisung könnten wir nun in unserem HTML-Markup durchführen:

```
<body onload="doStuff();">
```

### 3.1 Aufgaben beim Laden der Seite durchführen

47

Es ist aber auch möglich, dies im JavaScript-Code zu erledigen:

```
window.onload = doStuff;
```

Bei beiden Vorgehensweisen wird die Funktion ausgeführt, wenn die Seite geladen ist. Der Vorteil der zweiten Variante besteht darin, dass das Verhalten sauberer vom Markup getrennt wird.

#### Auf Funktionen verweisen oder sie direkt aufrufen

Wenn wir eine Funktion als Handler zuweisen, verwenden wir ihren Namen, lassen aber die angehängten Klammern weg. Bei Angabe der Klammern wird die Funktion sofort aufgerufen; ohne die Klammern *verweist* der Name einfach auf die Funktion und kann dazu verwendet werden, sie später aufzurufen.

Bei nur einer Funktion klappt diese Vorgehensweise sehr gut. Stellen Sie sich aber vor, wir hätten eine zweite Funktion:

```
function doOtherStuff() {  
    // Führt eine andere Aufgabe durch ...  
}
```

Wir könnten jetzt versuchen, diese Funktion zuzuweisen, sodass sie beim Laden der Seite ausgeführt wird:

```
window.onload = doOtherStuff;
```

Diese Zuweisung überschreibt aber die erste. Das Attribut `.onload` kann immer nur einen Funktionsverweis auf einmal speichern, weshalb wir diese Zuweisung nicht zu dem bereits bestehenden Verhalten hinzufügen können.

Der Mechanismus von `$(document).ready()` dagegen kann elegant mit dieser Situation umgehen. Jeder Aufruf der Methode fügt die neue Funktion der internen Warteschlange für Verhaltensweisen hinzu, und wenn die Seite geladen ist, werden sämtliche Funktionen ausgelöst und in der Reihenfolge ihrer Registrierung ausgeführt.

Der Fairness halber müssen wir darauf hinweisen, dass jQuery kein Monopol auf Lösungen für dieses Problem hat. Wir können eine JavaScript-Funktion schreiben, die den vorhandenen `onload`-Handler und dann einen übergebenen Handler auruft. Dieser Ansatz vermeidet ebenso wie `$(document).ready()` Konflikte zwischen konkurrierenden Handlern, allerdings fehlen ihm einige der anderen Vorteile, die wir besprochen haben. In modernen Browsern – einschließlich Internet Explorer 9 – kann das Ereignis `DOMContentLoaded` durch die standardmäßige W3C-Methode `document.addEventListener()` ausgelöst werden. jQuery aber kümmert sich auch um die Inkonsistenzen bei älteren Browsern, sodass wir das nicht selbst tun müssen, wenn wir solche Browser unterstützen wollen.

### 3.1.3 Kurzschrreibweisen

Das Konstrukt `$(document).ready()` ruft die Methode `.ready()` für das jQuery-Objekt auf, das wir aus dem DOM-Element `document` erstellt haben. Da dies eine häufig vorkommende Aufgabe ist, bietet die Funktion `$()` eine Kurzschrreibweise dafür. Wenn wir eine Funktion als Argument übergeben, ruft jQuery implizit `.ready()` auf. Um also dasselbe Ergebnis zu erzielen wie mit diesem Code:

```
$(document).ready(function() {
    // Unser Code ...
});
```

können wir auch Folgendes schreiben:

```
$(function() {
    // Unser Code ...
});
```

Die zweite Version ist zwar kürzer, doch die erste macht deutlicher, was der Code tut. Daher verwenden wir überall in diesem Buch die längere Syntax.

### 3.1.4 Argumente an den `.ready()-Callback` übergeben

In manchen Fällen ist es sinnvoll, mehr als eine JavaScript-Bibliothek auf derselben Seite zu verwenden. Da viele Bibliotheken den Bezeichner `$` benutzen (da er kurz und einfach einzugeben ist), brauchen wir eine Möglichkeit, um Konflikte zu vermeiden.

Zum Glück gibt es in jQuery die Methode `jQuery.noConflict()`, um die Kontrolle über den Bezeichner `$` an andere Bibliotheken zurückzugeben. Ein Beispiel für die typische Verwendung von `jQuery.noConflict()` finden Sie im folgenden Listing:

```
<script src="prototype.js"></script>
<script src="jquery.js"></script>
<script>
    jQuery.noConflict();
</script>
<script src="myscript.js"></script>
```

Als Erstes wird die andere Bibliothek (in diesem Beispiel Prototype) inkludiert. Danach wird jQuery eingeschlossen und übernimmt das `$` für den eigenen Gebrauch. Als Nächstes gibt der Aufruf von `.noConflict()` das `$` jedoch wieder frei, sodass die Kontrolle auf die zuerst eingeschlossene Bibliothek (Prototype) übergeht. In unserem Skript können wir jetzt beide Bibliotheken verwenden. Wann immer wir jetzt eine jQuery-Methode einsetzen, müssen wir als Bezeichner jedoch jQuery schreiben und nicht `$`.

Die Methode `.ready()` hat noch einen weiteren Trumpf im Ärmel, der uns in dieser Situation helfen kann. Die Callback-Funktion, die wir ihr übergeben haben, kann einen einzigen Parameter übernehmen, nämlich das jQuery-Objekt selbst. Dadurch können wir es ohne das Risiko von Konflikten umbenennen, wie der folgende Codeausschnitt zeigt:

```
jQuery(document).ready(function($) {  
    // Hier können wir das $ normal verwenden!  
});
```

In der kürzeren Syntax, die wir im vorherigen Code gelernt haben, sieht das wie folgt aus:

```
jQuery(function($) {  
    // Code, der $ verwendet.  
});
```

## 3.2 Einfache Ereignisse

Neben dem Laden der Seite gibt es viele andere Zeitpunkte, zu denen wir eine Aufgabe durchführen möchten. JavaScript ermöglicht uns nicht nur, das Ladeereignis der Seite mit `<body onload="">` oder `window.onload` abzufangen, sondern bietet auch ähnliche Hooks für vom Benutzer ausgelöste Ereignisse wie Mausklicks (`onclick`), die Änderung von Formularfeldern (`onchange`) und der Fenstergröße (`onresize`). Wenn diese Hooks direkt DOM-Elementen zugewiesen werden, ziehen sie ähnliche Nachteile nach sich wie diejenigen, die wir für `onload` besprochen haben. Daher bietet jQuery eine bessere Möglichkeit, um auch diese Ereignisse zu handhaben.

### 3.2.1 Ein einfacher Formatwechsler

Um einige Techniken der Ereignisbehandlung zu veranschaulichen, betrachten wir das Beispiel einer Seite, die je nach Benutzereingabe in verschiedenen Layoutformaten dargestellt wird. Der Benutzer kann auf Schaltflächen klicken, um zwischen drei Ansichten zu wechseln: der Normalansicht (DEFAULT), einer Ansicht, in der der Text auf eine schmale Spalte beschränkt ist (NARROW COLUMN), und einer Ansicht mit großer Schrift in den Inhaltsbereichen (LARGE PRINT).

#### Fortschreitende Verbesserung

In der Praxis würde ein guter Webbrowser hier das Prinzip der *fortschreitenden Verbesserung* umsetzen. Wenn JavaScript nicht verfügbar ist, sollte der Formatwechsler ausgeblendet sein oder, was noch besser wäre, mithilfe von Links zu alternativen Versionen der Seite trotzdem funktionieren. In diesem Lehrbeispiel jedoch setzen wir voraus, dass alle Benutzer JavaScript aktiviert haben.

Das HTML-Markup für den Formatwechsler sieht wie folgt aus:

```
<div id="switcher" class="switcher">
    <h3>Style Switcher</h3>
    <button id="switcher-default">
        Default
    </button>
    <button id="switcher-narrow">
        Narrow Column
    </button>
    <button id="switcher-large">
        Large Print
    </button>
</div>
```

Zusammen mit dem restlichen HTML-Markup der Seite und einer grundlegenden CSS-Gestaltung ergibt sich folgende Darstellung:



Als Erstes sorgen wir dafür, dass die Schaltfläche LARGE PRINT funktioniert. Um die alternative Ansicht der Seite zu realisieren, benötigen wir nur ein bisschen CSS-Code:

```
body.large .chapter {
    font-size: 1.5em;
}
```

Anschließend müssen wir die Klasse large dem <body>-Tag zuweisen, sodass das Stylesheet die Seite entsprechend umformatieren kann. Nach dem, was wir in Kapitel 2, »Elemente auswählen«, gelernt haben, können wir dies mit der folgenden Anweisung erreichen:

```
$(‘body’).addClass(‘large’);
```

Allerdings soll dies geschehen, wenn der Benutzer auf die Schaltfläche klickt, und nicht, wenn die Seite geladen wird, wie es bisher der Fall war. Dazu führen wir die Methode `.bind()` ein, mit der wir ein DOM-Ereignis spezifizieren und es mit einem Verhalten verknüpfen können. In diesem Fall ist das Ereignis `click` und das Verhalten eine Funktion, die aus unserem vorherigen Einzeiler besteht:

```
$(document).ready(function() {  
    $('#switcher-large').bind('click', function() {  
        $('body').addClass('large');  
    });  
});
```

**Listing 3–1**

Wenn jetzt jemand auf die Schaltfläche klickt, wird der Code ausgeführt, sodass der Text wie in der folgenden Abbildung größer dargestellt wird:

A screenshot of a web page for "A Christmas Carol" by Charles Dickens. At the top right, there is a "Style Switcher" button with three options: "Default", "Narrow Column", and "Large Print". The main content area displays the title "A Christmas Carol" and subtitle "In Prose, Being a Ghost Story of Christmas" by Charles Dickens. Below this is the "Preface" section, which begins with a quote: "I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it." It then continues with "Their faithful Friend and Servant," followed by "C. D." and the date "December, 1843.". The first chapter is titled "Stave I: Marley's Ghost", and the first sentence is "MARLEY was dead: to bain with. There is no doubt whatever".

Das ist auch schon das ganze Geheimnis der Bindung von Verhalten an Ereignisse. Die Vorteile, die wir schon für die Verwendung der Methode `.ready()` aufgeführt haben, gelten auch hier. Mehrere Aufrufe von `.bind()` können problemlos nebeneinander existieren, sodass Sie einem Ereignis bei Bedarf mehrere Verhalten hinzufügen können.

Dies ist jedoch nicht unbedingt die eleganteste oder effizienteste Möglichkeit, um die gegebene Aufgabe zu erledigen. Im weiteren Verlauf dieses Kapitels werden wir den Code erweitern und verfeinern, bis Sie stolz auf ihn sein können.

### 3.2.2 Die anderen Schaltflächen aktivieren

Die Schaltfläche LARGE PRINT funktioniert jetzt wie vorgesehen, aber wir müssen noch die beiden anderen Schaltflächen (DEFAULT und NARROW COLUMN) ähnlich handhaben, damit sie ihre Aufgaben erfüllen. Das geht ganz einfach: Wir verwenden jeweils `.bind()`, um einen click-Handler zu diesen Schaltflächen hinzuzufügen, und entfernen bzw. ergänzen Klassen nach Bedarf. Der Code sieht wie folgt aus:

```
$(document).ready(function() {
    $('#switcher-default').bind('click', function() {
        $('body').removeClass('narrow');
        $('body').removeClass('large');
    });
    $('#switcher-narrow').bind('click', function() {
        $('body').addClass('narrow');
        $('body').removeClass('large');
    });
    $('#switcher-large').bind('click', function() {
        $('body').removeClass('narrow');
        $('body').addClass('large');
    });
});
```

#### ***Listing 3–2***

Dazu kommt noch eine CSS-Regel für die Klasse narrow:

```
body.narrow .chapter {
    width: 250px;
}
```

Nach einem Klick auf NARROW COLUMN wird der zugehörige CSS-Code angewandt, sodass der Text ein anderes Layout erhält:

The screenshot shows a web page with a light gray background. In the top right corner, there is a rectangular button labeled "Style Switcher" with three rounded tabs below it: "Default", "Narrow Column", and "Large Print". The "Narrow Column" tab is currently highlighted. Below the button, the title "A Christmas Carol" and subtitle "In Prose, Being a Ghost Story of Christmas" are displayed in bold black font. Underneath, it says "by Charles Dickens". A section titled "Preface" is shown, followed by a paragraph of text: "I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it." At the bottom of this section, it says "Their faithful Friend and Servant," followed by two small circular icons.

Klickt der Benutzer auf DEFAULT, werden beide Klassennamen aus dem <body>-Tag entfernt, sodass die Seite zu ihrer ursprünglichen Darstellung zurückkehrt.

### 3.2.3 Ereignishandler-Kontext

Unser Formatwechsler verhält sich zwar korrekt, aber wir geben dem Benutzer keine Rückmeldung darüber, welche Schaltfläche gerade aktiv ist. Um das nachzuholen, fügen wir die Klasse `selected` einer Schaltfläche hinzu, wenn darauf geklickt wurde, und entfernen sie von den anderen Schaltflächen. Diese Klasse sorgt einfach dafür, dass die Beschriftung der Schaltfläche fett dargestellt wird:

```
.selected {  
    font-weight: bold;  
}
```

Diese Klassenänderung hätten wir auch durchführen können, indem wir jede Schaltfläche über ihre ID ansprechen und die Klassen nach Bedarf hinzufügen oder entfernen. Stattdessen aber sehen wir uns nun eine elegantere und anpassungsfähigere Lösung an, die den *Kontext* nutzt, in dem die Ereignishandler ausgeführt werden.

Wenn irgendein Ereignishandler ausgelöst wird, verweist das Schlüsselwort `this` auf das DOM-Element, mit dem das Verhalten verknüpft wurde. Wir haben bereits festgestellt, dass die Funktion `$( )` ein DOM-Element als Argument annehmen kann. Was wir hier tun, ist einer der Hauptgründe dafür, dass diese Möglichkeit vorgesehen wurde. Wenn wir innerhalb des Ereignishandlers `$(this)` schreiben, erstellen wir ein jQuery-Objekt, das dem Element entspricht, und können dieses Element daher bearbeiten, als hätten wir es mit einem CSS-Selektor ausgewählt.

Mit dieser Erklärung im Hinterkopf schreiben wir den folgenden Code:

```
$(this).addClass('selected');
```

Wenn wir diese Zeile in jeden der drei Handler einbauen, fügen wir die Klasse hinzu, wenn auf die Schaltfläche geklickt wird. Um die Klasse von den anderen Schaltflächen zu entfernen, nutzen wir die implizite Iteration von jQuery aus und schreiben folgenden Code:

```
$('#switcher button').removeClass('selected');
```

Diese Zeile entfernt die Klasse von jeder Schaltfläche innerhalb des Formatwechslers.

Wir müssen die Klasse auch zur Schaltfläche `DEFAULT` hinzufügen, wenn das Dokument bereitsteht. In der richtigen Reihenfolge angeordnet, ergibt sich folgender Code:

```
$(document).ready(function() {
    $('#switcher-default')
        .addClass('selected')
        .bind('click', function() {
            $('body').removeClass('narrow');
            $('body').removeClass('large');
            $('#switcher button').removeClass('selected');
            $(this).addClass('selected');
        });
    $('#switcher-narrow').bind('click', function() {
        $('body').addClass('narrow');
        $('body').removeClass('large');
        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
    });
    $('#switcher-large').bind('click', function() {
        $('body').removeClass('narrow');
        $('body').addClass('large');
        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
    });
});
```

***Listing 3–3***

Jetzt gibt der Formatwechsler eine angemessene Rückmeldung.

Die Generalisierung der Anweisungen durch Nutzung des Handler-Kontextes ermöglicht es uns sogar, noch effizienter vorzugehen. Wir können die Routine zum Hervorheben der Beschriftung wie in Listing 3–4 in einen eigenen Handler auslagern, da sie für alle drei Schaltflächen identisch ist:

```
$(document).ready(function() {
    $('#switcher-default')
        .addClass('selected')
        .bind('click', function() {
            $('body').removeClass('narrow').removeClass('large');
        });
    $('#switcher-narrow').bind('click', function() {
        $('body').addClass('narrow').removeClass('large');
    });
    $('#switcher-large').bind('click', function() {
        $('body').removeClass('narrow').addClass('large');
    });

    $('#switcher button').bind('click', function() {
        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
    });
});
```

***Listing 3–4***

Diese Optimierung nutzt die drei jQuery-Eigenschaften aus, wie wir bereits behandelt haben. Erstens nutzen wir wiederum die *implizite Iteration*, wenn wir denselben click-Handler mit einem einzigen Aufruf von .bind() an jede der Schaltflächen binden. Zweitens ermöglichen uns die *Verhaltenswarteschlangen*, zwei Funktionen an dasselbe Klickereignis zu binden, ohne dass die zweite die erste überschreibt. Drittens greifen wir auf die Möglichkeit der *Verkettung* in jQuery zurück, um das Hinzufügen und Entfernen von Klassen jedes Mal in einer einzigen Codezeile zusammenzufassen.

### 3.2.4 Weitere Konsolidierung

Die Codeoptimierung, die wir gerade durchgeführt haben, ist ein Beispiel für das *Refactoring* – die Veränderung bestehenden Codes, um dieselbe Aufgabe auf effizientere oder elegantere Weise zu erledigen. Um weitere Möglichkeiten zu einem Refactoring zu untersuchen, schauen wir uns die Verhalten an, die wir an die einzelnen Schaltflächen gebunden haben. Der Parameter der Methode .removeClass() ist optional. Wenn er weggelassen wird, werden alle Klassen von dem Element entfernt. Wir können unseren Code noch ein bisschen optimieren, indem wir diesen Umstand wie folgt ausnutzen:

```
// work in progress
$(document).ready(function() {
    $('#switcher-default')
        .addClass('selected')
        .bind('click', function() {
            $('body').removeClass();
        });
    $('#switcher-narrow').bind('click', function() {
        $('body').removeClass().addClass('narrow');
    });
    $('#switcher-large').bind('click', function() {
        $('body').removeClass().addClass('large');
    });

    $('#switcher button').bind('click', function() {
        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
    });
});
```

**Listing 3–5**

Die Reihenfolge der Operationen hat sich geändert, um der neuen Vorgehensweise Rechnung zu tragen, alle Klassen zu entfernen: Wir müssen zuerst .removeClass() ausführen, damit dadurch nicht die Wirkung von .addClass() rückgängig gemacht wird.

Wir können nur deshalb alle Klassen gefahrlos entfernen, weil wir die Kontrolle über den HTML-Code haben. Wenn wir Code schreiben, der zur Wiederverwendung gedacht ist (z.B. für ein Plug-in), müssen wir alle Klassen berücksichtigen, die vielleicht schon vorhanden sein könnten, und sie unberührt lassen.

Zurzeit führen wir in den Handlern der einzelnen Schaltflächen noch teilweise identischen Code aus. Auch diesen können wir auf einfache Weise in einen allgemeinen click-Handler für Schaltflächen auslagern, wie das folgende Listing zeigt:

```
$(document).ready(function() {
    $('#switcher-default').addClass('selected');

    $('#switcher button').bind('click', function() {
        $('body').removeClass();
        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
    });

    $('#switcher-narrow').bind('click', function() {
        $('body').addClass('narrow');
    });

    $('#switcher-large').bind('click', function() {
        $('body').addClass('large');
    });
});
```

**Listing 3–6**

Beachten Sie, dass wir den allgemeinen Handler vor den spezifischen platzieren müssen. Die Methode `.removeClass()` muss also vor `.addClass()` ausgeführt werden und wir können uns darauf verlassen, da jQuery die Ereignishandler immer in der Reihenfolge auslöst, in der sie registriert sind.

Zu guter Letzt können wir die spezifischen Handler vollständig loswerden, indem wir wiederum den *Ereigniskontext* ausnutzen. Da uns das Kontextschlüsselwort `this` Zugang zu einem DOM-Element bietet statt zu einem jQuery-Objekt, können wir die systemeigenen DOM-Eigenschaften nutzen, um die ID des Elements zu bestimmen, auf das geklickt wurde. Damit können wir denselben Handler an alle drei Schaltflächen binden und innerhalb des Handlers jeweils unterschiedliche Aktionen für die einzelnen Schaltflächen durchführen:

```
$(document).ready(function() {
    $('#switcher-default').addClass('selected');

    $('#switcher button').bind('click', function() {
        var bodyClass = this.id.split('-')[1];

        $('body').removeClass().addClass(bodyClass);
```

```
$('#switcher button').removeClass('selected');
$(this).addClass('selected');
});
});
```

**Listing 3–7**

Die Variable `bodyClass` nimmt, je nachdem, auf welche Schaltfläche geklickt wurde, die Werte `default`, `narrow` oder `large` an. Hier weichen wir ein wenig von unserem früheren Code ab, indem wir die Klasse `default` zu `<body>` hinzufügen, wenn der Benutzer auf `<button id="switcher-default">` klickt. Es ist zwar nicht notwendig, diese Klasse hinzuzufügen, aber es schadet auch nicht, und die Vereinfachung des Codes gleicht die Verwendung eines ungenutzten Klassennamens mehr als aus.

### 3.2.5 Kurzformen für Ereignisse

Handler an Ereignisse zu binden (z.B. an ein einfaches `click`-Ereignis), ist eine so häufige Aufgabe, dass jQuery eine noch knappere Möglichkeit dafür bietet. Die Kurzformen der Ereignismethoden funktionieren genauso wie die Gegenstücke mit `.bind()`, erfordern aber weniger Eingabearbeit.

Beispielsweise könnten wir unseren Formatwechsler mit `.click()` statt mit `.bind()` schreiben, wie der folgende Codeausschnitt zeigt:

```
$(document).ready(function() {
  $('#switcher-default').addClass('selected');

  $('#switcher button').click(function() {
    var bodyClass = this.id.split('-')[1];

    $('body').removeClass().addClass(bodyClass);

    $('#switcher button').removeClass('selected');
    $(this).addClass('selected');
  });
});
```

**Listing 3–8**

Ereignismethoden in Kurzfassung wie diese gibt es für alle standardmäßigen DOM-Ereignisse:

- blur
- change
- click
- dblclick
- error
- focus
- keydown

- keypress
- keyup
- load
- mousedown
- mousemove
- mouseout
- mouseover
- mouseup
- resize
- scroll
- select
- submit
- unload

Jede Kurzmethode bindet einen Handler an das Ereignis mit dem entsprechenden Namen.

### 3.3 Zusammengesetzte Ereignisse

Die meisten Ereignisbehandlungsmethoden von jQuery entsprechen direkt den systemeigenen DOM-Ereignissen. Es gibt jedoch auch eine Handvoll jQuery-eigener Handler, die aus praktischen Gründen und zur browserübergreifenden Optimierung hinzugefügt wurden. Einen davon, die Methode `.ready()`, haben wir bereits ausführlich besprochen. Andere, wie `.mouseenter()`, `.mouseleave()`, `.focusin()` und `.focusout()`, standardisieren proprietäre Internet Explorer-Ereignisse desselben Namens. Zwei der jQuery-Handler, nämlich `.toggle()` und `.hover()`, werden als *zusammengesetzte Ereignishandler* bezeichnet, da sie kombinierte Benutzeraktionen erfassen und mit mehr als einer Funktion darauf reagieren.

#### 3.3.1 Erweiterte Funktionen anzeigen und ausblenden

Nehmen wir an, wir wollen unseren Formatwechsler ausblenden, wenn er nicht gebraucht wird. Eine bequeme Möglichkeit zum Ausblenden von Zusatzfunktionen besteht darin, sie minimierbar zu gestalten: Mit einem Klick auf die Beschriftung werden die Schaltflächen ausgeblendet, sodass nur die Beschriftung zurückbleibt, und ein weiterer Klick auf die Beschriftung bringt die Schaltflächen zurück. Für die ausgeblendeten Schaltflächen brauchen wir eine neue Klasse:

```
.hidden {  
    display: none;  
}
```

Diese Funktionsweise könnten wir realisieren, indem wir den aktuellen Status der Schaltflächen in einer Variablen speichern und diesen Wert bei jedem Klick auf die Beschriftung prüfen, um zu bestimmen, ob wir die Klasse `hidden` zu den Schaltflächen hinzufügen oder sie davon entfernen müssen. Es wäre auch möglich, das Vorhandensein der Klasse in einer Schaltfläche direkt zu überprüfen und diese Information heranzuziehen, um zu entscheiden, was zu tun ist. Stattdessen aber bietet jQuery die Methode `.toggle()`, die diese Verwaltungsaufgaben für uns übernimmt.

#### Effekte beim Umschalten

In jQuery sind in Wirklichkeit zwei `.toggle()`-Methoden definiert. Informationen über die Effektmethode dieses Namens (die sich jeweils von der anderen durch andere Argumenttypen unterscheidet) finden Sie unter <http://api.jquery.com/toggle/>.

Die Ereignismethode `.toggle()` nimmt zwei Argumente an, die beide Funktionen sind. Beim ersten Klick auf das Element wird die erste Funktion ausgelöst, beim zweiten Klick die zweite usw. Sobald beide Funktionen aufgerufen worden sind, beginnt der Zyklus wieder mit der ersten. Mit `.toggle()` können wir unseren minimierbaren Formatwechsler auf ganz einfache Weise verwirklichen:

```
$(document).ready(function() {
    $('#switcher h3').toggle(function() {
        $('#switcher button').addClass('hidden');
    }, function() {
        $('#switcher button').removeClass('hidden');
    });
});
```

**Listing 3–9**

Nach dem ersten Klick sind alle Schaltflächen ausgeblendet:

The screenshot shows a web page with a light gray background. In the top right corner, there is a small rectangular button with a black border containing the white text "Style Switcher". Below this button, there is a large rectangular area with a thin black border. Inside this area, at the top left, is the bolded text "A Christmas Carol". Below it is the text "In Prose, Being a Ghost Story of Christmas" in a smaller bolded font. Underneath that, in a smaller regular font, is the text "by Charles Dickens". At the very bottom of this inner area, there is some extremely small, illegible text. The rest of the page is blank white space.

Nach dem zweiten Klick sind sie wieder sichtbar:



Auch hier verlassen wir uns wieder auf die implizite Iteration, in diesem Fall, um alle Schaltflächen in einem Rutsch auszublenden, ohne dafür ein Element zu benötigen, das sie alle einschließt.

In diesem besonderen Fall bietet jQuery noch einen anderen Mechanismus für die Minimierung. Mit der Methode `.toggleClass()` können wir automatisch überprüfen, ob die Klasse vorhanden ist, bevor wir sie hinzufügen oder entfernen:

```
$(document).ready(function() {
    $('#switcher h3').click(function() {
        $('#switcher button').toggleClass('hidden');
    });
});
```

#### ***Listing 3–10***

In diesem Beispiel stellt `.toggleClass()` wahrscheinlich die elegantere Lösung dar, doch `.toggle()` ist im Allgemeinen eine vielseitige Möglichkeit, um zwei oder mehrere verschiedene Aktionen im Wechsel durchzuführen.

### **3.3.2 Anklickbare Elemente hervorheben**

Um die Möglichkeiten zu illustrieren, die das `click`-Ereignis bietet, um Operationen an normalerweise nicht anklickbaren Seitenelementen durchzuführen, haben wir eine Oberfläche gestaltet, die kaum Hinweise darauf gibt, dass die Beschriftung des Formatwechslers – ein einfaches `<h3>`-Element – in Wirklichkeit ein aktiver Bestandteil der Seite ist und auf Benutzeraktionen wartet. Hier können wir Abhilfe schaffen, indem wir der Beschriftung einen Rollover-Status verleihen, um deutlich zu machen, dass sie in irgendeiner Weise auf die Maus reagiert:

```
.hover {
    cursor: pointer;
    background-color: #afa;
}
```

Die CSS-Spezifikation enthält die Pseudoklasse `:hover`, die es einem Stylesheet ermöglicht, das Erscheinungsbild eines Elements zu ändern, wenn der Mauszeiger darüberfährt. In Internet Explorer 6 ist diese Möglichkeit auf Links

beschränkt, sodass wir sie nicht für andere Elemente verwenden können, wenn wir diesen veralteten Browser unterstützen müssen. Vor allem aber wollen wir – nach dem Prinzip der *fortschreitenden Verbesserung* – das `<h3>`-Element nur dann als anklickbar gestalten, wenn wir es mit unserem jQuery-Code anklickbar gemacht haben. Zum Glück ermöglicht es uns die jQuery-Methode `.hover()`, mithilfe von JavaScript die Gestaltung eines Elements zu ändern – und auch jede andere mögliche Aktion daran auszuführen –, wenn der Mauszeiger über das Element fährt und wenn er wieder davon zurückgezogen wird.

Wie `.toggle()` in unserem vorhergehenden Beispiel nimmt auch die Methode `.hover()` zwei Funktionsargumente an. In diesem Fall wird die erste Funktion ausgeführt, wenn der Mauszeiger in den Bereich des ausgewählten Elements eintritt, und die zweite, wenn er den Bereich wieder verlässt. Um einen Rollover-Effekt zu erzielen, können wir wie folgt festlegen, welche Klassen den Schaltflächen zu diesen Zeitpunkten zugewiesen werden:

```
$(document).ready(function() {
    $('#switcher h3').hover(function() {
        $(this).addClass('hover');
    }, function() {
        $(this).removeClass('hover');
    });
});
```

**Listing 3–11**

Abermals nutzen wir die implizite Iteration und den Ereigniskontext aus, um kurzen, einfachen Code zu schreiben. Wenn wir jetzt mit dem Mauszeiger über die `<h3>`-Überschrift fahren, wird die entsprechende Klasse zugewiesen, wie wir in der folgenden Abbildung sehen:

The screenshot shows a web page with a header containing a 'Style Switcher' button with three options: 'Default', 'Narrow Column', and 'Large Print'. Below the header, the main content area displays the title 'A Christmas Carol' and the subtitle 'In Prose, Being a Ghost Story of Christmas' by Charles Dickens. It includes a 'Preface' section with text about the book's purpose and a note from C. D. dated December, 1843. The first chapter, 'Stave I: Marley's Ghost', begins with a short paragraph about Marley's death.

**A Christmas Carol**  
**In Prose, Being a Ghost Story of Christmas**  
by Charles Dickens

**Preface**

I HAVE endeavoured in this Ghostly little book, to raise the Ghost of an Idea, which shall not put my readers out of humour with themselves, with each other, with the season, or with me. May it haunt their houses pleasantly, and no one wish to lay it.

Their faithful Friend and Servant,  
C. D.  
December, 1843.

**Stave I: Marley's Ghost**

MARLEY was dead: to begin with. There is no doubt whatever about that. The register of his burial

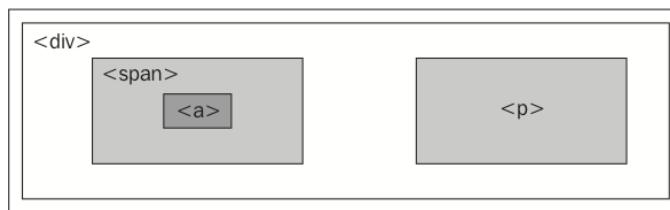
Die Verwendung von `.hover()` bewahrt uns auch davor, uns wegen der *Ereignisweiterleitung* in JavaScript Gedanken zu machen. Um dieses Problem zu verstehen, sehen wir uns an, wie JavaScript entscheidet, welches Element ein gegebenes Ereignis verarbeitet.

### 3.4 Der Weg eines Ereignisses

Wenn auf einer Seite ein Ereignis auftritt, steht eine ganze Hierarchie von DOM-Elementen bereit, um es zu verarbeiten. Stellen Sie sich ein Seitenmodell wie das folgende vor:

```
<div class="foo">
  <span class="bar">
    <a href="http://www.example.com/">
      The quick brown fox jumps over the lazy dog.
    </a>
  </span>
  <p>
    How razorback-jumping frogs can level six piqued gymnasts!
  </p>
</div>
```

Wenn wir uns den Code grafisch in Form geschachtelter Elemente vorstellen, ergibt sich folgende Abbildung:

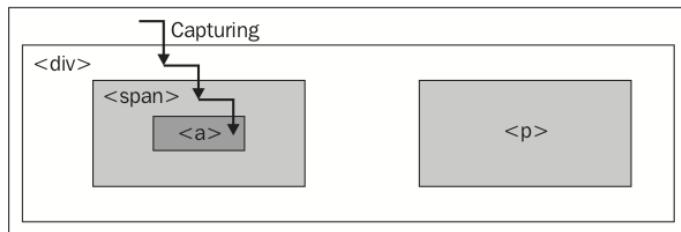


Bei jedem Ereignis gibt es mehrere Elemente, die logisch in Betracht kommen, darauf zu reagieren. Wenn beispielsweise ein Link auf der Seite angeklickt wird, sollten `<div>`, `<span>` und `<a>` die Gelegenheit haben, darauf zu reagieren, denn schließlich liegen alle drei Elemente zu diesem Zeitpunkt unter dem Mauszeiger. Das `<p>`-Element dagegen hat keinen Anteil an dieser Interaktion.

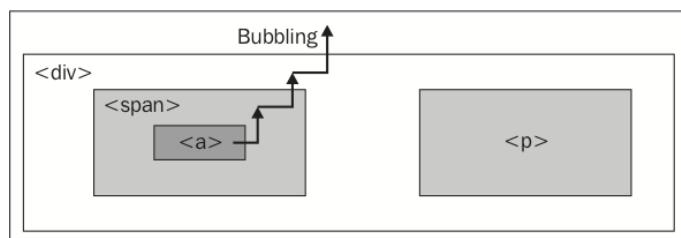
Eine Vorgehensweise, um die Reaktion mehrerer Elemente auf eine Benutzeraktion zuzulassen, ist das *Event Capturing* (das »Abfangen« von Ereignissen). Dabei wird das Ereignis zunächst dem alles umfassenden Element übergeben und dann nach und nach den spezifischeren. In unserem Beispiel heißt das, dass das

Ereignis erst an `<div>` übergeben wird, dann an `<span>` und schließlich an `<a>`, wie Sie in der folgenden Abbildung sehen:

Bei der technischen Umsetzung des Event Capturing in Browsern sind besondere Elemente dafür registriert, auf Ereignisse zu achten, die bei ihren Nachfahren auftreten. Die Verallgemeinerung in dieser Darstellung reicht für unsere Zwecke jedoch aus.



Die gegenteilige Vorgehensweise ist das *Event Bubbling* (»Ereignissprudeln«). Das Ereignis wird an das spezifischste Element gesendet, und nachdem dieses die Gelegenheit zur Reaktion bekommen hat, »sprudelt« das Ereignis förmlich nach oben zu den allgemeineren Elementen. In unserem Beispiel würde dabei `<a>` das Ereignis als Erstes verarbeiten, anschließend `<span>` und dann `<div>`, wie die folgende Abbildung zeigt:



Wie nicht anders zu erwarten, haben sich die verschiedenen Browserhersteller jeweils für unterschiedliche Modelle der Ereignisweiterleitung entschieden. Der DOM-Standard, der sich schließlich entwickelte, gibt daher an, dass beide Verfahren verwendet werden sollen: Erst wird das Element von den allgemeinen Elementen abgefangen und an die spezifischeren weitergereicht, und anschließend sprudelt es wieder zur Spitze des DOM-Baums empor. Für beide Teile dieses Vorgangs können Ereignishandler registriert werden.

Nicht alle Browser sind jedoch darauf aktualisiert worden, diesen neuen Standard umzusetzen, und in denjenigen, die das Event Capturing unterstützen, muss es gewöhnlich ausdrücklich aktiviert werden. Um für eine browserübergreifende Kontinuität zu sorgen, registriert jQuery Ereignishandler immer für die Bubbling-Phase des Modells. Wir können also immer davon ausgehen, dass die spezifischsten Elemente als erste die Gelegenheit haben, auf ein Ereignis zu reagieren.

### 3.4.1 Nebenwirkungen des Event Bubbling

Das Event Bubbling kann zu unvorhergesehenem Verhalten führen, vor allem wenn das falsche Element auf `mouseover` oder `mouseout` reagiert. Nehmen wir an, dass in unserem Beispiel ein `mouseout`-Ereignishandler mit dem `<div>` verknüpft ist. Wenn der Mauszeiger das `<div>` verlässt, wird der `mouseout`-Handler wie erwartet ausgeführt. Da dies an der Spitze der Hierarchie geschieht, bekommen die anderen Elemente das Ereignis nicht mit. Verlässt der Mauszeiger dagegen das `<a>`-Element, wird ein `mouseout`-Ereignis dorthin gesandt. Dieses Ereignis sprudelt zu `<span>` und `<div>` hoch und löst dort denselben Ereignishandler aus, was jedoch höchstwahrscheinlich nicht beabsichtigt ist.

Die Ereignisse `mouseenter` und `mouseleave`, die entweder einzeln oder in Kombination an die Methode `.hover()` gebunden sind, kennen diese Problematik. Wenn wir sie zur Verknüpfung mit Ereignissen verwenden, müssen wir uns keine Gedanken darüber machen, dass das falsche Element das `mouseover`- oder `mouseout`-Ereignis empfängt.

Das `mouseout`-Beispiel zeigt, dass wir den Gültigkeitsbereich eines Elements einschränken müssen. Zwar kümmert sich `.hover()` in diesem Sonderfall darum, doch werden wir anderen Situationen begegnen, in denen wir ein Ereignis räumlich eingrenzen müssen (damit es an bestimmte Elemente nicht gesendet wird) oder zeitlich (damit es zu bestimmten Zeitpunkten nicht gesendet wird).

## 3.5 Den Weg ändern: das Ereignisobjekt

Wir haben bereits eine Situation kennengelernt, in der das Event Bubbling Probleme bereiten kann. Um auch einen Fall vorzuführen, in dem uns `.hover()` nicht weiterhilft, ändern wir das Minimierungsverhalten, das wir zuvor codiert haben.

Nehmen wir an, wir wollen den anklickbaren Bereich ausdehnen, der die Minimierung oder Erweiterung des Formatwechslers auslöst. Eine Möglichkeit besteht darin, den Ereignishandler von der Beschriftung, dem `<h3>`-Element, zu dem umschließenden `<div>`-Element zu verschieben:

```
// Unfinished code
$(document).ready(function() {
    $('#switcher').click(function() {
        $('#switcher button').toggleClass('hidden');
    });
});
```

**Listing 3–12**

Durch diese Änderung wird die gesamte Fläche des Formatwechslers anklickbar, um die Sichtbarkeit umzuschalten. Der Nachteil besteht darin, dass bei einem Klick auf eine Schaltfläche nun auch der Formatwechsler minimiert wird, nachdem die Formatierung des Inhalts geändert wurde. Das liegt am Event Bubbling: Das Ereignis wird erst von den Schaltflächen verarbeitet und dann durch den DOM-Baum nach oben weitergereicht, bis es <div id="switcher"> erreicht, wo es unseren neuen Handler aktiviert und die Schaltflächen ausblendet.

Um dieses Problem zu lösen, müssen wir auf das *Ereignisobjekt* zugreifen. Dabei handelt es sich um ein DOM-Konstrukt, das den Ereignishandlern der einzelnen Elemente bei deren Aufruf übergeben wird. Es liefert Informationen über das Ereignis, z.B. wo sich der Mauszeiger zu dem betreffenden Zeitpunkt befand. Außerdem enthält es einige Methoden, die die Weiterleitung des Ereignisses durch das DOM beeinflussen können.

#### Informationen über das Ereignisobjekt

Ausführliche Informationen über die jQuery-Implementierung des Ereignisobjekts und seiner Eigenschaften finden Sie unter <http://api.jquery.com/category/events/event-object/>.

Um in unseren Handlern das Ereignisobjekt verwenden zu können, müssen wir der Funktion lediglich einen Parameter hinzufügen:

```
$(document).ready(function() {
    $('#switcher').click(function(event) {
        $('#switcher button').toggleClass('hidden');
    });
});
```

Beachten Sie, dass wir diesen Parameter nur deshalb `event` genannt haben, weil dies eine anschauliche Beschreibung ist, und nicht etwa, weil es erforderlich wäre. Wir hätten ihn auch `haferkeks` nennen können, und es würde genauso gut funktionieren.

### 3.5.1 Ereignisziele

Das Ereignisobjekt steht uns jetzt in Form der Variablen `event` in unserem Handler zur Verfügung. Die Eigenschaft `event.target` ist hilfreich für die Steuerung, wo ein Ereignis in Kraft tritt. Diese Eigenschaft gehört zur DOM-API, ist aber nicht in allen Browsern implementiert. Um sie in jedem Browser bereitzustellen, erweitert jQuery bei Bedarf das Ereignisobjekt. Mit `.target` können wir bestimmen, welches Element im DOM das Ereignis als Erstes empfangen hat. Im Fall eines `click`-Ereignisses ermitteln wir damit also, auf welches Element geklickt wurde. Indem wir ausnutzen, dass uns `this` das DOM-Element gibt, das das Ereignis verarbeitet, können wir folgenden Code schreiben:

```
// Unfinished code
$(document).ready(function() {
    $('#switcher').click(function(event) {
        if (event.target == this) {
            $('#switcher button').toggleClass('hidden');
        }
    });
});
```

**Listing 3-13**

Dieser Code stellt sicher, dass das Element, auf das geklickt wurde, `<div id="switcher">` war und nicht eines seiner Unterelemente. Ein Klick auf eine der Schaltflächen führt jetzt nicht mehr dazu, dass der Formatwechsler minimiert wird, wie dies beim Klick auf den Hintergrund des Wechslers geschieht. Allerdings passiert jetzt auch bei einem Klick auf die Beschriftung, das `<h3>`-Element, nichts, denn auch dies ist ein Unterelement. Anstatt hier ebenfalls eine Überprüfung durchzuführen, ändern wir das Verhalten der Schaltflächen, um unser Ziel zu erreichen.

### 3.5.2 Die Ereignisweiterleitung abbrechen

Das Ereignisobjekt besitzt die Methode `.stopPropagation()`, die das »Hochsprudeln« der Ereignisse komplett zum Stillstand bringen kann. Wie `.target` entstammt auch diese Methode dem normalen JavaScript, kann aber ebenfalls nicht gefahrlos in allen Browsern verwendet werden. Solange wir all unsere Ereignishandler jedoch mit jQuery registrieren, können wir sie ungestraft einsetzen.

Wir entfernen jetzt die Überprüfung `event.target == this`, die wir gerade eingefügt haben, und ergänzen dafür den Code in den `click`-Handlern unserer Schaltflächen wie folgt:

```
$(document).ready(function() {
    $('#switcher').click(function(event) {
        $('#switcher button').toggleClass('hidden');
    });
});
```

```
$(document).ready(function() {
    $('#switcher-default').addClass('selected');

    $('#switcher button').click(function(event) {
        var bodyClass = this.id.split('-')[1];

        $('body').removeClass().addClass(bodyClass);

        $('#switcher button').removeClass('selected');
        $(this).addClass('selected');
        event.stopPropagation();
    });
});
```

**Listing 3–14**

Wie zuvor müssen wir der Funktion, die wir als click-Handler benutzen, einen Parameter hinzufügen, um Zugriff auf das Ereignisobjekt zu bekommen. Dann rufen wir einfach event.stopPropagation() auf, um zu verhindern, dass irgendein anderes DOM-Element auf das Ereignis reagiert. Jetzt wird der Klick von den Schaltflächen verarbeitet, und zwar nur von den Schaltflächen. Klicks, die an anderer Stelle im Formatwechsler stattfinden, führen dazu, dass der Wechsler minimiert oder erweitert wird.

### 3.5.3 Standardaktionen

Wenn wir unsere click-Ereignishandler für ein Linkelement (<a>) statt für einen allgemeinen <button> außerhalb eines Formulars registrieren, stehen wir vor einem anderen Problem: Klickt ein Benutzer auf den Link, lädt der Browser eine neue Seite. Dieses Verhalten ist kein Ereignishandler in dem zuvor besprochenen Sinne, sondern eine *Standardaktion* für den Klick auf ein Linkelement. Ebenso wird das Ereignis submit für ein Formular ausgelöst, wenn der Benutzer während der Bearbeitung die Enter-Taste drückt, woraufhin das Formular übertragen wird.

Wenn diese Standardaktionen nicht erwünscht sind, hilft es nichts, .stopPropagation() für das Ereignis aufzurufen, denn diese Aktionen treten nicht im normalen Verlauf der Ereignisweiterleitung auf. In diesen Fällen muss die Methode .preventDefault() dazu verwendet werden, das Ereignis im Keim zu ersticken, bevor die Standardaktion ausgelöst wird.

Ein Aufruf von .preventDefault() ist oft sinnvoll, nachdem wir einige Tests in der Umgebung des Ereignisses vorgenommen haben. Beispielsweise können wir bei der Übertragung eines Formulars überprüfen, ob die erforderlichen Felder ausgefüllt sind, und die Standardaktion nur dann unterbinden, wenn das nicht der Fall ist.

Ereignisweiterleitung und Standardaktionen sind voneinander unabhängig. Jeder dieser Mechanismen kann angehalten werden, während der andere weiterhin abläuft. Wenn wir beide anhalten wollen, können wir am Ende unseres Ereignishandlers `false` zurückgeben. Das ist die Kurzform für den Aufruf von sowohl `.stopPropagation()` als auch `.preventDefault()` für das Ereignis.

### 3.5.4 Ereignisdelegierung

Das Event Bubbling ist nicht immer eine Hindernis, sondern kann auch große Vorteile bringen. Eine hervorragende Technik, die sich darauf stützt, ist die *Ereignisdelegierung*. Damit können wir einen Ereignishandler für ein einziges Element dazu verwenden, um die Arbeit von vielen anderen zu erledigen.

In unserem Beispiel gibt es nur drei `<button>`-Elemente, mit denen wir `click`-Handler verknüpft haben. Was aber, wenn es viel mehr wären? So etwas kommt häufiger vor, als man denkt. Stellen Sie sich beispielsweise eine umfangreiche Tabelle vor, in der jede Zeile über ein interaktives Element verfügt, für das ein `click`-Handler erforderlich ist. Die implizite Iteration macht es zwar einfach, all diese `click`-Handler zuzuweisen, aber die Performance kann unter der internen Schleifenverarbeitung in jQuery und dem Speicherverbrauch für all die Handler leiden.

Stattdessen können wir einen einzigen `click`-Handler zu einem Vorfahrenelement im DOM zuweisen. Ein `click`-Ereignis, dessen Weiterleitung nicht unterbrochen wird, erreicht dank des Event Bubbling schließlich diesen Vorfahren, sodass wir die Arbeit dort erledigen können.

Zur Veranschaulichung wenden wir diese Technik bei unserem Formatwechsler an (auch wenn die Anzahl der Elemente diese Vorgehensweise nicht notwendig macht). Wie wir in Listing 3–13 gesehen haben, können wir mit der Eigenschaft `event.target` prüfen, welches Element sich bei dem Klick unter dem Mauszeiger befand:

```
$(document).ready(function() {
    $('#switcher').click(function(event) {
        if ($(event.target).is('button')) {
            var bodyClass = event.target.id.split('-')[1];
            $('body').removeClass().addClass(bodyClass);

            $('#switcher button').removeClass('selected');
            $(event.target).addClass('selected');
            event.stopPropagation();
        }
    });
});
```

**Listing 3–15**

Hier haben wir eine neue Methode verwendet, nämlich `.is()`. Sie nimmt die Selektorausdrücke an, die wir uns im vorangegangenen Kapitel angesesehen haben, und prüft das aktuelle jQuery-Objekt auf diesen Selektor. Wenn mindestens ein Element in der Menge mit dem Selektor übereinstimmt, gibt `.is()` den Wert `true` zurück. In diesem Fall prüft `$(event.target).is('button')`, ob es sich bei dem angeklickten Element um ein `<button>`-Element handelt. Wenn ja, fahren wir mit dem bereits bekannten Code fort, wobei es jedoch eine wesentliche Änderung gibt: Das Schlüsselwort `this` verweist jetzt auf `<div id="switcher">`, weshalb wir, wann immer wir an der angeklickten Schaltfläche interessiert sind, mit `event.target` darauf verweisen müssen.

#### `.is()` und `.hasClass()`

Das Vorhandensein einer Klasse in einem Element können wir mit `.hasClass()` überprüfen. Die Methode `.is()` ist jedoch vielseitiger und kann einen Test mit jedem beliebigen Selektorausdruck durchführen.

Dieser Code ruft jedoch einen unbeabsichtigten Seiteneffekt hervor. Bei einem Klick auf eine Schaltfläche wird jetzt wieder der Formatwechsler minimiert, wie es vor der Ergänzung um `.stopPropagation()` der Fall war. Der Handler für den Minimierungsumschalter ist jetzt an dasselbe Element gebunden wie der Handler für die Schaltflächen, weshalb das Anhalten des Event Bubbling nicht dazu führt, dass das Auslösen der Minimierung unterbunden wird. Um dieses Problem zu vermeiden, können wir den Aufruf von `.stopPropagation()` entfernen und stattdessen einen weiteren `.is()`-Test einfügen.

Da wir das gesamte `<div>`-Element des Formatwechslers anklickbar gemacht haben, müssen wir außerdem die Klasse ändern, wenn sich die Maus über irgendeinem Bereich davon befindet, wie es im folgenden Code geschieht:

```
$(document).ready(function() {
    $('#switcher').hover(function() {
        $(this).addClass('hover');
    }, function() {
        $(this).removeClass('hover');
    });
});

$(document).ready(function() {
    $('#switcher').click(function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    });
});
```

```
$(document).ready(function() {
    $('#switcher-default').addClass('selected');

    $('#switcher').click(function(event) {
        if ($(event.target).is('button')) {
            var bodyClass = event.target.id.split('-')[1];

            $('body').removeClass().addClass(bodyClass);

            $('#switcher button').removeClass('selected');
            $(event.target).addClass('selected');
        }
    });
});
```

**Listing 3–16**

Diese Vorgehensweise ist für eine Anwendung dieser Größe zwar übermäßig kompliziert, aber je mehr Elemente mit Ereignishandlern es gibt, umso vorteilhafter wird die Ereignisdelegierung. Außerdem können wir einen Teil der Codewiederholung vermeiden, indem wir zwei click-Handler kombinieren und für den .is()-Test eine einzige if-else-Anweisung verwenden:

```
$(document).ready(function() {
    $('#switcher-default').addClass('selected');

    $('#switcher').click(function(event) {
        if ($(event.target).is('button')) {
            var bodyClass = event.target.id.split('-')[1];

            $('body').removeClass().addClass(bodyClass);

            $('#switcher button').removeClass('selected');
            $(event.target).addClass('selected');
        } else {
            $('#switcher button').toggleClass('hidden');
        }
    });
});
```

**Listing 3–17**

Unser Code braucht zwar immer noch einen gewissen Feinschliff, aber wir nähern uns einem Zustand, in dem wir ihn reinen Gewissens für unseren Zweck einsetzen können. Um mehr über die Ereignisbehandlung in jQuery zu lernen, kehren wir jedoch zu Listing 3–16 zurück und ändern diese Version des Codes.

Wie wir noch sehen werden, ist die Ereignisdelegierung auch in anderen Situationen nützlich, z.B. wenn neue Elemente durch Methoden zur *DOM-Bearbeitung* (siehe Kapitel 5, »DOM-Bearbeitung«) oder durch *Ajax-Routinen* (siehe Kapitel 6, »Daten mit Ajax senden«) hinzugefügt werden.

### 3.5.5 Methoden für die Ereignisdelegierung

Da die Ereignisdelegierung in vielen Situationen hilfreich sein kann, enthält jQuery eine Reihe von Methoden eigens für diese Technik. In Kapitel 10, »Komplexe Ereignisse«, sehen wir uns diese Methoden (`.live()`, `.die()`, `.delegate()` und `.undelegate()`) noch genauer an. Schon an dieser Stelle möchten wir aber darauf hinweisen, dass `.live()` als Ersatz für `.bind()` dienen kann und dabei viele der Vorteile der Ereignisdelegierung bietet. Um beispielsweise einen aktiven `click`-Handler an die Schaltflächen des Formatwechslers zu binden, können wir folgenden Code schreiben:

```
$('#switcher button').live('click', function() {  
    var bodyClass = event.target.id.split('-')[1];  
  
    $('body').removeClass().addClass(bodyClass);  
  
    $('#switcher button').removeClass('selected');  
    $(this).addClass('selected');  
});
```

Hinter den Kulissen bindet jQuery den `click`-Handler an das `document`-Objekt und prüft mit `event.target`, ob es (oder irgendeiner seiner Vorfahren) mit dem Selektorausdruck übereinstimmt, in diesem Fall mit `'#switcher button'`. Wenn ja, ordnet jQuery das Schlüsselwort `this` dem passenden Element zu. Bei dieser Vorgehensweise lässt sich zwar die Auswahl mehrerer Elemente nicht vermeiden, doch wird trotzdem verhindert, dass eine Bindung an alle diese Elemente erfolgt. Da das `document`-Objekt selbst zur Verfügung steht, und zwar auch dann, wenn auf das Skript im `<head>`-Tag verwiesen wird, können `.live()`-Ereignisse überdies immer außerhalb von `$(document).ready()` gebunden werden.

## 3.6 Ereignishandler entfernen

Es kann vorkommen, dass wir einen zuvor registrierten Ereignishandler nicht mehr brauchen, z.B. weil sich der Zustand der Seite so geändert hat, dass die betreffende Aktion nicht mehr sinnvoll wäre. Meistens ist es möglich, diese Situation mit herkömmlichen Anweisungen innerhalb der Ereignishandler zu handhaben, aber es kann eleganter sein, die Bindung des Handlers vollständig aufzuheben.

Nehmen wir an, unser minimierbarer Formatwechsler soll immer erweitert bleiben, wenn die Seite nicht im normalen Format dargestellt wird. Wenn also die Schaltfläche NARROW COLUMN oder LARGE PRINT aktiv ist, sollte bei einem Klick auf den Hintergrund des Formatwechslers nichts geschehen. Das können wir erreichen, indem wir die Methode `.unbind()` aufrufen, um den Minimierungshandler zu entfernen, wenn auf eine Schaltfläche für die nicht standardmäßigen Formate geklickt wurde:

```
$(document).ready(function() {
    $('#switcher').click(function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    });
    $('#switcher-narrow, #switcher-large').click(function() {
        $('#switcher').unbind('click');
    });
});
```

**Listing 3–18**

Wenn der Benutzer jetzt z.B. auf die Schaltfläche NARROW COLUMN klickt, wird der click-Handler im <div>-Element des Formatwechslers entfernt, sodass das Feld bei einem Klick auf seinen Hintergrund nicht mehr minimiert wird. Aber jetzt funktionieren auch die Schaltflächen nicht mehr! Das click-Ereignis des <div>-Elements wirkt sich auch auf sie aus, da wir den Code zur Verarbeitung der Schaltflächen umgeschrieben haben, um die Ereignisdelegierung zu nutzen. Wenn wir \$('#switcher').unbind('click') aufrufen, werden daher beide Verhalten entfernt.

### 3.6.1 Namensräume für Ereignisse

Wir müssen den Aufruf von .unbind() präzisieren, damit dadurch nicht beide registrierte Ereignishandler entfernt werden. Eine Möglichkeit dazu bieten *Ereignis-Namensräume*. Beim Binden eines Ereignisses können wir zusätzliche Informationen angeben, um diesen Handler später genau zu identifizieren. Um Namensräume zu verwenden, müssen wir zu der ausführlichen Methode zur Bindung von Ereignishandlern zurückkehren, nämlich zur Methode .bind().

Als ersten Parameter übergeben wir .bind() den Namen des Ereignisses, auf das wir achten möchten. Dazu können wir hier jedoch eine besondere Syntax verwenden, die es uns ermöglicht, das Ereignis in eine Unterkategorie einzuteilen, wie der folgende Code zeigt:

```
$(document).ready(function() {
    $('#switcher').bind('click.collapse', function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    });
    $('#switcher-narrow, #switcher-large').click(function() {
        $('#switcher').unbind('click.collapse');
    });
});
```

**Listing 3–19**

Das Suffix `.collapse` ist für das Ereignisbehandlungssystem nicht sichtbar. Diese Funktion verarbeitet `click`-Ereignisse genauso, als hätten wir `.bind('click')` geschrieben. Durch das Hinzufügen eines Namensraumes können wir jedoch diesen einen Handler entfernen, ohne dass sich dies auf die anderen `click`-Handler auswirkt, die wir für die Schaltflächen geschrieben haben.

Wie wir gleich sehen werden, gibt es noch andere Möglichkeiten, um den Aufruf von `.unbind()` präziser zu machen. Namensräume für Ereignisse sind jedoch ein nützliches Instrument. Vor allem beim Erstellen von Plug-ins sind sie sehr praktisch, wie wir in späteren Kapiteln erfahren werden.

### 3.6.2 Ereignisse erneut binden

Ein Klick auf die Schaltflächen `LARGE PRINT` oder `NARROW COLUMN` führt jetzt dazu, dass die Funktion zum Minimieren des Formatwechslers deaktiviert wird. Allerdings soll dieses Verhalten nach einem Klick auf `DEFAULT` wiederhergestellt werden. Dazu müssen wir den Handler *neu binden*, wenn der Benutzer auf `DEFAULT` klickt.

Als Erstes geben wir dazu unserer Handlerfunktion einen Namen, sodass wir sie mehr als einmal benutzen können, ohne den Code wiederholen zu müssen:

```
$(document).ready(function() {
    var toggleSwitcher = function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    };
    $('#switcher').bind('click.collapse', toggleSwitcher);
});
```

**Listing 3-20**

Beachten Sie die neue Syntax für die Definition einer Funktion, die wir hier verwenden. Anstelle einer *Funktionsdeklaration* (einer Definition der Funktion durch das Schlüsselwort `function` mit darauffgendem Funktionsnamen) verwenden wir hier einen *anonymen Funktionsausdruck*, indem wir eine namenlose Funktion zu einer *lokalen Variablen* zuweisen. Neben einigen feinen Unterschieden, die in diesem Fall keine Rolle spielen, sind die beiden Schreibweisen funktional gleichwertig. Hier haben wir den Funktionsausdruck aus stilistischen Gründen gewählt, damit unsere Ereignishandler und die anderen Funktionsdefinitionen einander stärker ähneln.

Denken Sie daran, dass `.bind()` einen *Funktionsverweis* als zweites Argument annimmt. Es ist wichtig, bei der Verwendung von benannten Funktionen die Klammern hinter dem Funktionsnamen wegzulassen, da sonst die Funktion als solche aufgerufen und nicht nur auf sie verwiesen wird.

Da jetzt auf die Funktion verwiesen werden kann, können wir sie später erneut binden, ohne die Funktionsdefinition wiederholen zu müssen:

```
// Unfinished code
$(document).ready(function() {
    var toggleSwitcher = function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    };
    $('#switcher').bind('click.collapse', toggleSwitcher);
    $('#switcher-narrow, #switcher-large').click(function() {
        $('#switcher').unbind('click.collapse');
    });
    $('#switcher-default').click(function() {
        $('#switcher')
            .bind('click.collapse', toggleSwitcher);
    });
});
```

**Listing 3-21**

Das Minimierungsverhalten wird jetzt beim Laden des Dokuments gebunden. Später wird die Bindung aufgehoben, wenn der Benutzer auf NARROW COLUMN oder LARGE PRINT klickt, und wiederhergestellt, wenn anschließend auf DEFAULT geklickt wird.

Da wir die Funktion benannt haben, benötigen wir den Namensraum nicht mehr. Die Methode `.unbind()` kann als zweites Argument eine Funktion annehmen und hebt in diesem Fall nur die Bindung des jeweiligen Handlers auf. Allerdings haben wir uns ein anderes Problem aufgehalst. Wenn ein Handler in jQuery an ein Ereignis gebunden wird, bleiben vorherige Handler in Kraft. In diesem Fall wird bei jedem Klick auf DEFAULT eine weitere Kopie des Handlers `toggleSwitcher` an den Formatwechsler gebunden. Mit anderen Worten, die Funktion wird bei jedem weiteren Klick erneut aufgerufen, bis der Benutzer auf NARROW COLUMN oder LARGE PRINT klickt, woraufhin die Bindung aller `toggleSwitcher`-Handler auf einmal aufgehoben wird.

Wenn eine gerade Anzahl von `toggleSwitcher`-Handlern gebunden sind, scheint ein Klick auf das Feld des Formatwechslers (nicht auf die Schaltflächen) keine Wirkung zu haben. Tatsächlich wird die Klasse `hidden` mehrmals umgeschaltet und hat am Ende wieder den Zustand, in dem sie sich zu Anfang befand. Um

dieses Problem zu lösen, können wir die Bindung des Handlers aufheben, wenn der Benutzer auf irgendeine Schaltfläche klickt, und die Bindung nur dann wiederherstellen, wenn wir sichergestellt haben, dass die ID der angeklickten Schaltfläche `switcher-default` lautete.

```
$(document).ready(function() {
    var toggleSwitcher = function(event) {
        if (!$(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    };
    $('#switcher').bind('click', toggleSwitcher);

    $('#switcher button').click(function() {
        $('#switcher').unbind('click', toggleSwitcher);

        if (this.id == 'switcher-default') {
            $('#switcher').bind('click', toggleSwitcher);
        }
    });
});
```

**Listing 3-22**

Für Situationen, in denen wir die Bindung eines Ereignishandlers aufheben wollen, unmittelbar nachdem er zum ersten Mal ausgelöst wurde, gibt es die Kurzschreibweise `.one()`, die wie folgt eingesetzt wird:

```
$('#switcher').one('click', toggleSwitcher);
```

Dadurch erfolgt die Umschaltung nur einmal.

## 3.7 Benutzerinteraktion simulieren

Manchmal ist es praktisch, den Code, den wir an ein Ereignis gebunden haben, ausführen zu können, auch wenn das betreffende Ereignis gar nicht eingetreten ist. Nehmen Sie beispielsweise an, dass wir unseren Formatwechsler so gestalten möchten, dass er zu Anfang in minimiertem Zustand angezeigt wird. Dazu könnten wir die Schaltflächen im Stylesheet ausblenden, unsere Klassen `hidden` hinzufügen oder die Methode `.hide()` von einem `$(document).ready()`-Handler aus aufrufen. Eine andere Möglichkeit besteht darin, einen Klick auf den Formatwechsler zu simulieren, damit der Umschaltmechanismus ausgelöst wird.

Das erreichen wir mit der Methode `.trigger()`:

```
$(document).ready(function() {
    $('#switcher').trigger('click');
});
```

**Listing 3-23**

Wenn die Seite jetzt geladen wird, ist der Wechsler minimiert, als hätte jemand darauf geklickt:



Wenn wir Inhalte ausblenden wollen, die Benutzer ohne aktiviertes JavaScript sehen sollen, dann ist dies eine sinnvolle Möglichkeit, um diese *allmähliche Funktionsverminderung* zu verwirklichen.

Für die Methode `.trigger()` werden dieselben Kurzformen verwendet wie für `.bind()`. Werden sie ohne Argumente eingesetzt, wird die zugehörige Aktion ausgelöst, anstatt sie zu binden:

```
$(document).ready(function() {
    $('#switcher').click();
});
```

**Listing 3–24**

### 3.7.1 Tastaturereignisse

Als weiteres Beispiel fügen wir Tastenkürzel zu unserem Formatwechsler hinzu. Wenn der Benutzer den ersten Buchstaben eines der Anzeigeformate eingibt, soll sich die Seite so verhalten, als ob er auf die entsprechende Schaltfläche geklickt hätte. Um diese Funktionsweise umzusetzen, müssen wir uns mit *Tastaturereignissen* beschäftigen, die sich ein wenig anders verhalten als *Mausereignisse*.

Es gibt zwei Arten von Tastaturereignissen: diejenigen, die direkt auf Tastenbewegungen reagieren (`keyup` und `keydown`), und diejenigen, die auf Texteingabe reagieren (`keypress`). Die Eingabe eines einzelnen Zeichens kann mit mehreren Tasten verbunden sein. Beispielsweise wird der Großbuchstabe X durch die kombinierte Verwendung der Tasten `Shift` und `X` erzeugt. Die Einzelheiten der Implementierung unterscheiden sich zwar von einem Browser zum anderen (was nicht weiter überraschend ist), doch als sichere Faustregel können wir Folgendes festhalten: Wenn Sie wissen wollen, welche Taste der Benutzer gedrückt hat, beobachten Sie das Ereignis `keyup` oder `keydown`, doch wenn Sie bestimmen müssen, welches Zeichen dadurch auf dem Bildschirm hervorgerufen wurde, müssen Sie sich `keypress` ansehen. Für das von uns beabsichtigte Verhalten müssen wir nur wissen, ob der Benutzer die Taste `D`, `N` oder `L` gedrückt hat, weshalb wir `keyup` verwenden.

Als Nächstes müssen wir festlegen, welches Element wir auf dieses Ereignis hin überwachen sollen. Das ist weniger offensichtlich als bei den Mausereignissen, bei denen uns der Mauszeiger Informationen über das Ziel des Ereignisses gibt. Das Ziel eines Tastaturereignisses dagegen ist das Element, auf dem zurzeit der *Tastaturfokus* liegt. Der Fokus kann auf verschiedene Weise von einem Element zum nächsten verschoben werden, z.B. durch Mausklicks oder mithilfe der Taste . Außerdem kann nicht jedes Element den Fokus bekommen, sondern nur Elemente mit tastaturgesteuertem Verhalten. Mögliche Kandidaten sind etwa Formularfelder, Links und Elemente mit der Eigenschaft `.tabIndex`.

In unserem Beispiel kümmern wir uns nicht darum, welches Element den Fokus hat. Unser Formatwechsler soll einfach reagieren, wenn der Benutzer eine der drei möglichen Tasten drückt. Auch hier erweist sich das Event Bubbling als nützlich, da wir das keyup-Ereignis an das document-Element binden und uns darauf verlassen können, dass jegliche Tastaturereignisse schließlich dorthin hochsprudeln werden.

Nun müssen wir noch wissen, welche Taste gedrückt wurde, wenn unser keyup-Handler ausgelöst wird. Dazu können wir das event-Objekt untersuchen. Die Eigenschaft `.keyCode` des Ereignisses enthält einen Bezeichner für die gedrückte Taste. Bei Buchstabentasten ist dies der ASCII-Wert des betreffenden Großbuchstabens. Wenn der Benutzer eine Taste drückt, schauen wir nach, ob es für diesen Bezeichner eine Zuordnung gibt. Ist das der Fall, lösen wir das entsprechende click-Ereignis aus.

```
$(document).ready(function() {
    var triggers = {
        D: 'default',
        N: 'narrow',
        L: 'large'
    };
    $(document).keyup(function(event) {
        var key = String.fromCharCode(event.keyCode);
        if (key in triggers) {
            $('#switcher-' + triggers[key]).click();
        }
    });
});
```

**Listing 3-25**

Das Drücken einer dieser drei Tasten simuliert jetzt einen Mausklick auf die zugehörige Schaltfläche – vorausgesetzt, das Tastaturereignis wird nicht durch Funktionen wie die »Textsuche während der Eingabe« von Firefox unterbrochen.

Anstatt den Klick mit `.trigger()` zu simulieren, können wir auch versuchen, den Code in eine Funktion auszulagern, sodass mehr als ein Handler dieses Verhalten aufrufen kann – in diesem Fall also sowohl `click` als auch `keyup`. Diese Technik hilft Coderedundanzen zu vermeiden, auch wenn das in diesem Fall nicht so auffällig ist:

```
$(document).ready(function() {
    // Aktiviert den Hover-Effekt für den Formatwechsler
    $('#switcher').hover(function() {
        $(this).addClass('hover');
    }, function() {
        $(this).removeClass('hover');
    });

    // Ermöglicht die Erweiterung und Minimierung des Formatwechslers
    var toggleSwitcher = function(event) {
        if (!(event.target).is('button')) {
            $('#switcher button').toggleClass('hidden');
        }
    };
    $('#switcher').bind('click', toggleSwitcher);

    // Simuliert einen Klick, sodass wir im minimierten Zustand beginnen
    $('#switcher').click();

    // Die Funktion setBodyClass() ändert das Seitenformat
    // Außerdem wird der Status des Formatwechslers aktualisiert
    var setBodyClass = function(className) {
        $('body').removeClass().addClass(className);

        $('#switcher button').removeClass('selected');
        $('#switcher-' + className).addClass('selected');

        $('#switcher').unbind('click', toggleSwitcher);

        if (className == 'default') {
            $('#switcher').bind('click', toggleSwitcher);
        }
    };

    // Beginnt mit aktiver Schaltfläche switcher-default
    $('#switcher-default').addClass('selected');

    // Ordnet die Tastencodes den zugehörigen Schaltflächen zu
    var triggers = {
        D: 'default',
        N: 'narrow',
        L: 'large'
    };
}
```

```
// Ruft beim Klick auf eine Schaltfläche setBodyClass() auf
$('#switcher').click(function(event) {
    if ($(event.target).is('button')) {
        var bodyClass = event.target.id.split('-')[1];
        setBodyClass(bodyClass);
    }
});

// Ruft beim Drücken einer Taste setBodyClass() auf
$(document).keyup(function(event) {
    var key = String.fromCharCode(event.keyCode);
    if (key in triggers) {
        setBodyClass(triggers[key]);
    }
});
});
```

**Listing 3–26**

## 3.8 Zusammenfassung

Mit den Möglichkeiten, die wir in diesem Kapitel behandelt haben, können wir Folgendes tun:

- Die Methode `.ready()` mit `.noConflict()` einsetzen, um auf einer Seite gleichzeitig mehrere JavaScript-Bibliotheken verwenden zu können.
- Mit *Mausereignishandlern* und den Methoden `.bind()` und `.click()` darauf reagieren, dass Benutzer auf Elemente der Seite klicken.
- Den *Ereigniskontext* beobachten, um je nach angeklicktem Element unterschiedliche Aktionen durchzuführen, selbst wenn der Handler an mehrere Elemente gebunden ist.
- Ein Seitenelement mit `.toggle()` abwechselnd erweitern und minimieren.
- Mit `.stopPropagation()` und `.preventDefault()` Einfluss auf die *Ereignisweiterleitung* und die *Standardaktionen* nehmen, um zu bestimmen, welche Elemente auf ein Ereignis reagieren.
- Eine *Ereignisdelegierung* durchführen, um die Anzahl der erforderlichen gebundenen Ereignishandler auf einer Seite zu verringern.
- Die Methode `.unbind()` aufrufen, um einen Ereignishandler zu entfernen, wenn wir ihn nicht mehr benötigen.
- Zusammengehörige Ereignishandler durch *Namensräume* gruppieren, sodass es möglich ist, sie als Gruppe zu behandeln.
- Die Ausführung gebundener Ereignishandler mit `.trigger()` auslösen.
- *Tastaturereignishandler* verwenden, um mit `.keyup()` darauf zu reagieren, dass der Benutzer eine Taste drückt.

Mit diesen Möglichkeiten können wir schon Seiten erstellen, die eine gewisse Interaktion bieten. Im nächsten Kapitel sehen wir uns an, wie wir dem Benutzer bei dieser Interaktion eine optische Rückmeldung geben.

### 3.8.1 Literatur

Das Thema Ereignisbehandlung wird ausführlicher in Kapitel 10 besprochen. Eine vollständige Liste der Ereignismethoden von jQuery finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## 3.9 Übungsaufgaben

Um die folgenden Übungen durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Wenden Sie das Format `selected` auf CHARLES DICKENS an, wenn der Benutzer auf diesen Namen klickt.
2. Schalten Sie die Sichtbarkeit des Kapiteltextes um, wenn der Benutzer doppelt auf eine Kapitelüberschrift klickt (`<h3> class="chapter-title">`).
3. Wechseln Sie beim Drücken der rechten Pfeiltaste zyklisch zur jeweils nächsten Body-Klasse. Der Code für diese Taste lautet 39.
4. Schwierig: Protokollieren Sie die Koordinaten des Mauszeigers bei der Bewegung über einen Absatz mit der Funktion `console.log()`. (Hinweis: `console.log()` zeigt seine Ereignisse in der Erweiterung Firebug für Firefox, im Web Inspector von Safari und in den Developer Tools von Chrome an.)
5. Schwierig: Verfolgen Sie mithilfe von `.mousedown()` und `.mouseup()` Mauseereignisse an beliebiger Stelle auf der Seite. Wenn die Maustaste oberhalb der Stelle losgelassen wird, an der sie gedrückt wurde, fügen Sie allen Paragraphen die Klasse `hidden` hinzu. Wird sie weiter unten losgelassen, entfernen Sie diese Klasse von allen Absätzen.

## 4 Formatierung und Animation

Taten mögen mehr sagen als Worte, aber in JavaScript sagen Effekte noch mehr als Taten (Aktionen). Mit jQuery können wir Aktionen mithilfe einer Reihe simpler optischer *Effekte* auf einfache Weise betonen. Es ist sogar möglich, eigene anspruchsvolle *Animationen* zu gestalten.

Die von jQuery angebotenen Effekte sorgen für eine einfache grafische Ausschmückung, um einer Seite einen dynamischen und modernen Anstrich zu geben. Sie sind allerdings nicht nur schmückendes Beiwerk, sondern können auch wichtige Verbesserungen darstellen, indem sie dem Benutzer eine Orientierungshilfe geben, wenn sich etwas auf einer Seite ändert (was vor allem in Ajax-Anwendungen üblich ist). In diesem Kapitel sehen wir uns einige dieser Effekte an und kombinieren sie auf interessante Weise.

### 4.1 Inline-Bearbeitung mit CSS

Bevor wir uns um die schicken jQuery-Effekte kümmern, werfen wir noch einen kurzen Blick auf CSS. In den ersten Kapiteln haben wir das Erscheinungsbild eines Dokuments geändert, indem wir Formate für Klassen in einem getrennten Stylesheet definiert und dann mithilfe von jQuery hinzugefügt oder entfernt haben. Das ist die übliche Vorgehensweise, um CSS in HTML-Code einzufügen, da hierbei deutlich wird, dass das Stylesheet für die Darstellung der Seite verantwortlich ist. Es kann jedoch vorkommen, dass wir Formate anwenden müssen, die noch nicht in einem Stylesheet definiert wurden oder die gar nicht so einfach darin definiert werden können. Für solche Gelegenheiten bietet jQuery die Methode `.css()`.

Diese Methode dient sowohl als *Get-* als auch als *Set-Methode*. Um den Wert einer Formateigenschaft abzurufen, übergeben wir ihren Namen einfach als String, z.B. `.css('backgroundColor')`. Eigenschaften, deren Namen aus mehreren Wörtern bestehen, kann jQuery interpretieren, wenn sie in der CSS-Schreibweise

mit Bindestrich (`background-color`) oder in der DOM-Schreibweise mit Binnenmajuskel (»camel case«) vorliegen (`backgroundColor`). Zur Festlegung von Formateigenschaften gibt es zwei Varianten der Methode `.css()`: Die eine benötigt als Parameter eine einzelne Formateigenschaft und deren Wert, die andere eine Zuordnung (*Map*) von Eigenschaft-Wert-Paaren, beides sehen Sie im folgenden Codeausschnitt:

```
// Einzelne Eigenschaft mit Wert
.css('property', 'value')

// Zuordnung von Eigenschaft-Wert-Paaren
.css({
  property1: 'value1',
  'property-2': 'value2'
})
```

Erfahrene JavaScript-Entwickler erkennen in diesen jQuery-Zuordnungen die *Objektliterale* von JavaScript.

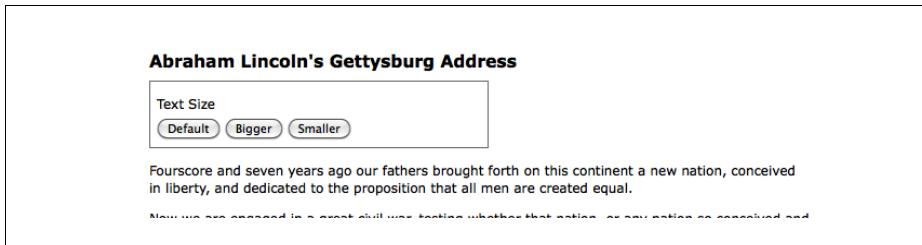
### Schreibweise von Objektliteralen

Als Eigenschaftswert sind Strings wie üblich in Anführungszeichen eingeschlossen, doch für andere Datentypen wie Zahlen ist das nicht erforderlich. Da Eigenschaftsnamen Strings sind, sollten sie eigentlich in Anführungszeichen stehen. Allerdings werden die Anführungszeichen nicht gebraucht, wenn die Eigenschaftsnamen gültige JavaScript-Bezeichner sind, was z.B. der Fall ist, wenn sie in der DOM-Schreibweise mit Binnenmajuskel vorliegen.

Die Methode `.css()` verwenden wir genauso wie `.addClass()`: Wir wenden sie auf ein jQuery-Objekt an, das wiederum auf eine Sammlung von DOM-Elementen zeigt. Dies demonstrieren wir mit einem ähnlichen Formatuschalter wie dem aus Kapitel 3, »Ereignisbehandlung«:

```
<div id="switcher">
  <div class="label">Text Size</div>
    <button id="switcher-default">Default</button>
    <button id="switcher-large">Bigger</button>
    <button id="switcher-small">Smaller</button>
</div>
<div class="speech">
  <p>Four score and seven years ago our fathers brought forth
    on this continent a new nation, conceived in liberty,
    and dedicated to the proposition that all men are created
    equal.
  </p>
</div>
```

Durch die Verknüpfung mit einem Stylesheet, das über wenige grundlegende Formatregeln verfügt, sieht die Seite zu Anfang wie in der folgenden Abbildung aus:



Wenn wir unseren Code fertiggestellt haben, sorgt ein Klick auf die Schaltflächen BIGGER und SMALLER dafür, dass die Textgröße von `<div class="speech">` vergrößert bzw. verringert wird, während der Text in diesem Bereich bei einem Klick auf DEFAULT auf die ursprüngliche Größe zurückgesetzt wird.

Wenn wir die Schriftgröße lediglich auf einen zuvor festgelegten Wert setzen wollten, können wir `.addClass()` verwenden. Wir gehen aber davon aus, dass die Textgröße bei jedem Klick auf die Schaltfläche inkrementell vergrößert oder verringert werden soll. Es wäre zwar möglich, für jeden Klick eine eigene Klasse zu definieren und dadurch zu iterieren, ein einfacherer Ansatz besteht aber darin, jedes Mal die neue Textgröße zu berechnen, indem wir die aktuelle Größe abrufen und sie um einen festen Faktor erhöhen (z.B. um 40%).

Unser Code beginnt wie folgt mit den Ereignishandlern `$(document).ready()` und `$('#switcher-large').click()`:

```
$(document).ready(function() {
  $('#switcher-large').click(function() {
    });
});
```

**Listing 4-1**

Danach können wir die Schriftgrößen mithilfe der Methode `.css()` ganz einfach ermitteln: `($('div.speech').css('fontSize'))`. Der zurückgegebene Wert ist jedoch ein String, der sowohl den numerischen Wert der Schriftgröße als auch die Einheit (px) enthält. Um die Berechnung mit dem numerischen Wert durchführen zu können, müssen wir die Einheit entfernen. Wenn wir das jQuery-Objekt mehr als einmal verwenden möchten, ist es außerdem im Allgemeinen sinnvoll, den Selektor *zwischenzuspeichern*, indem wir das resultierende jQuery-Objekt in einer Variablen ablegen. Dazu führen wir zwei lokale Variablen ein:

```
$(document).ready(function() {
    var $speech = $('div.speech');
    $('#switcher-large').click(function() {
        var num = parseFloat($speech.css('fontSize'));
    });
});
```

**Listing 4–2**

Die erste Zeile innerhalb von `$(document).ready()` erstellt jetzt eine Variable mit einem jQuery-Objekt, das auf `<div class="speech">` zeigt. Beachten Sie das `$`-Zeichen im Variablennamen `$speech`. Da `$` in JavaScript-Bezeichnern ein gültiges Zeichen ist, können wir es als Erinnerungsstütze dafür verwenden, dass wir in dieser Variablen ein jQuery-Objekt speichern.

Innerhalb des `.click()`-Handlers verwenden wir `parseFloat()`, um nur den numerischen Wert der Eigenschaft für die Schriftgröße abzurufen. Die Funktion `parseFloat()` untersucht einen String von links nach rechts, bis sie auf ein nicht numerisches Zeichen stößt, und verwandelt den Ziffernstring dann in eine (dezimale) Fließkommazahl. Beispielsweise würde aus dem String '12' die Zahl 12. Außerdem entfernt die Funktion alle nachfolgenden nicht numerischen Zeichen aus dem String, sodass aus '12px' ebenfalls 12 wird. Wenn der String mit einem nicht numerischen Zeichen beginnt, gibt `parseFloat()` den Wert `Nan` zurück, was *not a number* bedeutet, also »keine Zahl«.

Jetzt müssen wir nur noch den numerischen Wert ändern und die Schriftgröße auf der Grundlage dieses neuen Werts anpassen. In unserem Beispiel erhöhen wir die Schriftgröße jedes Mal um 40%, wenn auf die Schaltfläche geklickt wird. Dazu multiplizieren wir `num` mit 1,4 und legen die Schriftgröße fest, indem wir `num` und 'px' verketten, wie der folgende Codeausschnitt zeigt:

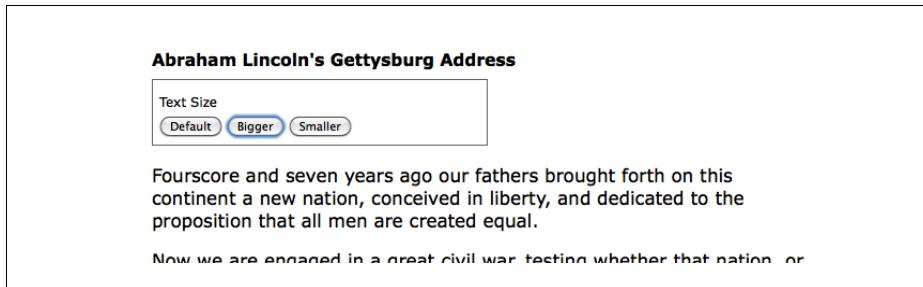
```
$(document).ready(function() {
    var $speech = $('div.speech');
    $('#switcher-large').click(function() {
        var num = parseFloat($speech.css('fontSize'));
        num *= 1.4;
        $speech.css('fontSize', num + 'px');
    });
});
```

**Listing 4–3**

**Abkürzungen für Operatoren**

Die Gleichung `num *= 1.4` ist die Abkürzung für `num = num * 1.4`. Diese Kurzfassung können wir auch für die anderen mathematischen Grundoperationen verwenden: die Addition (`a += b`), die Subtraktion (`a -= b`), die Division (`a /= b`) und den Modulus- oder Restoperator (`a %= b`).

Bei einem Klick auf BIGGER wird der Text jetzt größer und beim nächsten Klick noch größer, wie Sie im folgenden Screenshot sehen:



Damit die Schaltfläche SMALLER die Schriftgröße verringert, müssen wir dividieren, anstatt zu multiplizieren: `num /= 1.4`. Es wird aber noch besser: Wir kombinieren die beiden Berechnungen in einem einzigen `.click()`-Handler für alle `<button>`-Elemente in `<div id="switcher">`. Nachdem wir den numerischen Wert herausgefunden haben, multiplizieren oder dividieren wir ihn je nach der ID der angeklickten Schaltfläche, wie das folgende Listing zeigt:

```
$(document).ready(function() {
    var $speech = $('div.speech');
    $('#switcher button').click(function() {
        var num = parseFloat($speech.css('fontSize'));
        if (this.id == 'switcher-large') {
            num *= 1.4;
        } else if (this.id == 'switcher-small') {
            num /= 1.4;
        }
        $speech.css('fontSize', num + 'px');
    });
});
```

#### **Listing 4-4**

In Kapitel 3 haben wir gelernt, dass wir auf die Eigenschaft `id` des DOM-Elements zugreifen können, auf das `this` verweist. Hier nutzen wir dies in den `if-` und `else-if`-Anweisungen. In diesem Fall ist es effizienter, `this` zu verwenden, als ein jQuery-Objekt zu erstellen, nur um den Wert einer Eigenschaft zu prüfen.

Es wäre auch schön, wenn wir die Schriftgröße auf den ursprünglichen Wert zurücksetzen könnten. Um dem Benutzer diese Möglichkeit zu bieten, müssen wir die Schriftgröße einfach in einer Variablen speichern, sobald das DOM bereitsteht. Dann können wir diesen Wert wiederherstellen, wenn der Benutzer auf `DEFAULT` klickt. Mit einer weiteren `else-if`-Anweisung könnten wir diesen Klick verarbeiten, doch eine `switch`-Anweisung wie im folgenden Code mag hier geeigneter sein:

```
$(document).ready(function() {
    var $speech = $('div.speech');
    var defaultSize = $speech.css('fontSize');
    $('#switcher button').click(function() {
        var num = parseFloat($speech.css('fontSize'));
        switch (this.id) {
            case 'switcher-large':
                num *= 1.4;
                break;
            case 'switcher-small':
                num /= 1.4;
                break;
            default:
                num = parseFloat(defaultSize);
        }
        $speech.css('fontSize', num + 'px');
    });
});
```

**Listing 4–5**

Wir überprüfen nach wie vor den Wert von `this.id` und ändern die Schriftgröße anhand des Ergebnisses, aber wenn der Wert weder `'switcher-large'` noch `'switcher-small'` ist, kehren wir mit `default` zur ursprünglichen Schriftgröße zurück.

## 4.2 Anzeigen und Verbergen

Die grundlegenden Methoden `.hide()` und `.show()` – ohne Parameter – können Sie sich als intelligente Kurzformen für `.css('display', 'string')` vorstellen, wobei `'string'` der entsprechende Anzeigewert ist. Der Effekt besteht darin, dass ein Satz an ausgewählten Elementen sofort und ohne Animation verborgen bzw. angezeigt wird.

Die Methode `.hide()` setzt das *Inline-Formatattribut* der Menge ausgewählter Elemente auf `display: none`. Das Intelligente daran ist, dass sich die Methode den Wert der Eigenschaft `display` merkt – gewöhnlich `block` oder `inline` –, bevor sie ihn in `none` ändert. Die Methode `.show()` setzt die Anzeigeeigenschaft der Menge ausgewählter Elemente dann auf den Wert zurück, den sie vor der Anwendung von `display: none` hatte.

### Die Eigenschaft `display`

Weitere Informationen über die Eigenschaft `display` und darüber, welche Auswirkung ihre Werte auf die Darstellung einer Webseite haben, erhalten Sie im Mozilla Developer Center unter <https://developer.mozilla.org/en/CSS/display/>. Beispiele sind unter <https://developer.mozilla.org/samples/cssref/display.html> zu finden.

Diese Fähigkeit von `.show()` und `.hide()` ist besonders dann hilfreich, wenn Sie Elemente verbergen, deren standardmäßige `display`-Eigenschaft in einem Style-sheet überschrieben wurde. Beispielsweise hat das Element `<li>` standardmäßig die Eigenschaft `display: block`, doch können wir das für ein horizontales Menü in `display: inline` ändern. Bei der Verwendung von `.show()` wird ein verborgenes Element wie dieses `<li>`-Tag glücklicherweise nicht einfach auf den Standardwert `display: block` zurückgesetzt, wodurch es in eine eigene Zeile rutschen würde, sondern in den vorherigen Zustand zurückversetzt, also zu `display: inline`. Das horizontale Design bleibt also erhalten.

Um diese beiden Methoden zu veranschaulichen, können wir sie auf den zweiten Absatz anwenden und im HTML-Code des Beispiels einen Link zum Weiterlesen (READ MORE) hinter dem ersten Absatz einfügen:

```
<div class="speech">
    <p>Four score and seven years ago our fathers brought forth
        on this continent a new nation, conceived in liberty,
        and dedicated to the proposition that all men are
        created equal.
    </p>
    <p>Now we are engaged in a great civil war, testing whether
        that nation, or any nation so conceived and so dedicated,
        can long endure. We are met on a great battlefield of
        that war. We have come to dedicate a portion of that
        field as a final resting-place for those who here gave
        their lives that the nation might live. It is altogether
        fitting and proper that we should do this. But, in a
        larger sense, we cannot dedicate, we cannot consecrate,
        we cannot hallow, this ground.
    </p>
    <a href="#" class="more">read more</a>
</div>
```

Wenn das DOM bereitsteht, wählen wir ein Element aus und rufen `.hide()` dafür auf:

```
$(document).ready(function() {
    $('p').eq(1).hide();
});
```

**Listing 4–6**

Die Methode `.eq()` ähnelt der Pseudoklasse `:eq()` aus Kapitel 2, »Elemente auswählen«. Sie gibt ein jQuery-Objekt zurück, das auf ein einzelnes Element mit dem angegebenen Index zeigt (wobei die Indexnummerierung bei null beginnt). In diesem Fall wählt die Methode den zweiten Absatz aus und verbirgt ihn. Das Ergebnis sieht folgendermaßen aus:

### Abraham Lincoln's Gettysburg Address

Text Size

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

[read more](#)

The brave men, living and dead, who struggled here have consecrated it, far above our poor power  
to add or detract. The world will little note, nor long remember what we say here, but it can never

Wenn der Benutzer am Ende des ersten Abschnitts auf READ MORE klickt, wird der Link verborgen und dafür der zweite Absatz angezeigt:

```
$(document).ready(function() {  
    $('p').eq(1).hide();  
    $('a.more').click(function() {  
        $('p').eq(1).show();  
        $(this).hide();  
        return false;  
    });  
});
```

#### ***Listing 4-7***

Beachten Sie die Verwendung von `return false`, um die Standardaktion des Links zu unterbinden. Die Rede wird jetzt wie folgt dargestellt:

### Abraham Lincoln's Gettysburg Address

Text Size

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that the nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow, this ground.

The brave men, living and dead, who struggled here have consecrated it, far above our poor power

Die Methoden `.hide()` und `.show()` sind schnell und nützlich, aber ziemlich unauffällig. Um sie ein bisschen aufzupolieren, können wir ihnen etwas Speed verleihen. Was es damit auf sich hat, erklären wir im nächsten Abschnitt.

## 4.3 Effekte und Speed

Wenn wir zu `.show()` oder `.hide()` etwas *Speed* hinzufügen (genauer gesagt eine *Dauer*), werden diese Methoden animiert – sie laufen in einem festgelegten Zeitraum ab. Die Methode `.hide('speed')` beispielsweise verringert gleichzeitig die Höhe, Breite und Deckkraft eines Elements, bis alle den Wert null haben. Zu diesem Zeitpunkt wird dann die CSS-Regel `display: none` angewendet. Die Methode `.show('speed')` vergrößert die Höhe des Elements von oben nach unten und die Breite von links nach rechts und erhöht die Deckkraft von 0 bis 1, bis der Inhalt komplett sichtbar ist.

### 4.3.1 Anzeigen mit »Geschwindigkeit«

Bei jedem jQuery-Effekt können wir die Standardgeschwindigkeiten `'slow'`, `'normal'` und `'fast'` verwenden. Bei `.show('slow')` wird der Effekt in 0,6 s abgeschlossen, bei `.show('normal')` in 0,4 s und bei `.show('fast')` in 0,2 s. Für eine höhere Genauigkeit können wir auch die gewünschte Anzahl an Millisekunden angeben, z.B. `.slow(850)`. Beachten Sie, dass wir im letzten Fall einen numerischen Wert angeben und daher keine Anführungszeichen verwenden.

Für die Anzeige des zweiten Absatzes von Lincolns »Gettysburg Address« werden wir unserem Beispiel eine Geschwindigkeit hinzufügen:

```
$(document).ready(function() {
    $('p').eq(1).hide();
    $('a.more').click(function() {
        $('p').eq(1).show('slow');
        $(this).hide();
        return false;
    });
});
```

**Listing 4-8**

Wenn wir mitten im Ablauf dieses Effekts einen Screenshot vom Erscheinen des zweiten Absatzes machen, sieht das Ergebnis wie folgt aus:

### Abraham Lincoln's Gettysburg Address

Text Size  
[Default](#) [Bigger](#) [Smaller](#)

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that

The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced.

~~It is rather for us to be here dedicated to the great task remaining before us—that from these~~

#### 4.3.2 Ein- und ausblenden

Die animierten Methoden `.show()` und `.hide()` sind sicherlich auffällig, doch animieren sie mehr Eigenschaften, als in der Praxis sinnvoll ist. Zum Glück gibt es in jQuery zwei weitere vorgefertigte Animationen, um einen subtileren Effekt zu erzeugen. Beispielsweise können wir mit `.fadeIn('slow')` den ganzen Absatz einfach dadurch erscheinen lassen, dass wir die Deckkraft nach und nach erhöhen:

```
$(document).ready(function() {
    $('p').eq(1).hide();
    $('a.more').click(function() {
        $('p').eq(1).fadeIn('slow');
        $(this).hide();
        return false;
    });
});
```

**Listing 4–9**

Wenn wir uns jetzt den Absatz ansehen, während der Effekt abläuft, erkennen wir Folgendes:

### Abraham Lincoln's Gettysburg Address

Text Size  
[Default](#) [Bigger](#) [Smaller](#)

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to dedicate a portion of that field as a final resting-place for those who here gave their lives that the nation might live. It is altogether fitting and proper that we should do this. But, in a larger sense, we cannot dedicate, we cannot consecrate, we cannot hallow, this ground.

The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never

Der Unterschied besteht darin, dass `.fadeIn()` als Erstes die Abmessungen des Absatzes festlegt, sodass sein Inhalt einfach eingeblendet werden kann. Um die Deckkraft nach und nach zu verringern, können wir `.fadeOut()` verwenden.

#### 4.3.3 Auseinander- und zusammenfalten

Die Einblendeffekte eignen gut für Elemente, die sich außerhalb des Dokumentflusses befinden. Beispielsweise werden sie gewöhnlich für »Leuchtkästen« verwendet, die der Seite überlagert werden. Bei einem Element, das zum Dokumentfluss gehört, kann der Aufruf von `.fadeIn()` jedoch dazu führen, dass das Dokument »springt«, um Platz für das neue Element zu schaffen, was nicht sehr ansprechend aussieht.

In diesen Fällen sind die jQuery-Methoden `.slideDown()` und `.slideUp()` häufig die richtige Wahl. Diese Effekte animieren nur die Höhe der ausgewählten Elemente. Damit unser Absatz vertikal auseinandergefaltet wird, rufen wir wie folgt `.slideDown('slow')` auf:

```
$(document).ready(function() {
    $('p').eq(1).hide();
    $('a.more').click(function() {
        $('p').eq(1).slideDown('slow');
        $(this).hide();
        return false;
    });
});
```

**Listing 4–10**

Wenn wir den Absatz jetzt auf halbem Wege der Animation betrachten, sehen wir Folgendes:

**Abraham Lincoln's Gettysburg Address**

Text Size

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived and so dedicated, can long endure. We are met on a great battlefield of that war. We have come to

The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note nor long remember what we say here, but it can never

Um den Effekt umzukehren, rufen wir `.slideUp()` auf.

#### 4.3.4 Zusammengesetzte Effekte

Manchmal müssen wir die Sichtbarkeit von Elementen umschalten, anstatt sie nur einmal anzuzeigen, wie wir es in den vorstehenden Beispielen getan haben. Dieses Umschalten können wir dadurch erreichen, dass wir zunächst die Sichtbarkeit der ausgewählten Elemente überprüfen und dann die geeignete Methode aufrufen. Im Folgenden verwenden wir wieder den Einblendeffekt, um das Beispielskript so zu ändern:

```
$(document).ready(function() {
    var $firstPara = $('p').eq(1);
    $firstPara.hide();
    $('a.more').click(function() {
        if ($firstPara.is(':hidden')) {
            $firstPara.fadeIn('slow');
            $(this).text('read less');
        } else {
            $firstPara.fadeOut('slow');
            $(this).text('read more');
        }
        return false;
    });
});
```

**Listing 4-11**

Wie schon in den früheren Kapiteln speichern wir den Selektor hier zwischen, damit wir das DOM nicht wiederholt durchlaufen müssen. Beachten Sie, dass wir den angeklickten Link nicht mehr verbergen, sondern stattdessen seinen Text ändern.

Um den Text in einem Element zu untersuchen und zu ändern, verwenden wir die Methode `.text()`, die ausführlicher in Kapitel 5, »DOM-Bearbeitung«, erläutert wird.

Eine `if-else`-Anweisung ist eine völlig korrekte Möglichkeit, um die Sichtbarkeit eines Elements umzuschalten, doch die *zusammengesetzten Effekte* von jQuery erlauben es uns, einen Teil der Bedingungslogik aus unserem Code zu entfernen. jQuery enthält die Methode `.toggle()`, die wie `.show()` und `.hide()` funktioniert und ebenso wie sie mit und ohne Geschwindigkeitsargument eingesetzt werden kann. Andere zusammengesetzte Methoden sind `.fadeToggle()` und `.slideToggle()`, die Elemente mit den entsprechenden Effekten anzeigen und verbergen. Der folgende Code zeigt, wie unser Skript aussieht, wenn wir die Methode `.slideToggle()` verwenden:

```
$(document).ready(function() {
    var $firstPara = $('p').eq(1);
    $firstPara.hide();
    $('a.more').click(function() {
        $firstPara.slideToggle('slow');
        var $link = $(this);
        if ($link.text() == 'read more') {
            $link.text('read less');
        } else {
            $link.text('read more');
        }
        return false;
    });
});
```

**Listing 4-12**

Um die Wiederholung von `$(this)` zu vermeiden und die Performance und Lesbarkeit zu erhöhen, speichern wir das Ergebnis in der Variablen `$link`. Da wir die bedingte Anweisung jetzt nur noch verwenden, um den Text zu ändern, prüft sie außerdem den Text des Links und nicht die Sichtbarkeit des zweiten Absatzes.

## 4.4 Benutzerdefinierte Animationen erstellen

Neben den vorgefertigten Effektmethoden gibt es in jQuery die leistungsfähige Methode `.animate()`, mit der wir eigene benutzerdefinierte Animationen erstellen und genau steuern können. Es gibt zwei Formen dieser Methode, wobei die erste bis zu vier Argumente annimmt:

1. Eine *Zuordnung (Map)* von Formateigenschaften und Werten, ähnlich wie die zuvor in diesem Kapitel besprochene Zuordnung von `.css()`.
2. Eine optionale *Geschwindigkeit*, die in Form eines vordefinierten Strings oder als Anzahl von Millisekunden angegeben wird.
3. Einen optionalen *Beschleunigungstyp*, eine erweiterte Option, die wir in Kapitel 11, »Anspruchsvolle Effekte«, besprechen werden.
4. Eine optionale *Callback-Funktion*, um die wir uns weiter hinten in diesem Kapitel kümmern werden.

Zusammengenommen sehen diese vier Argumente aus wie in dem folgenden Codeausschnitt:

```
.animate({property1: 'value1', property2: 'value2'},
    speed, easing, function() {
        alert('The animation is finished.');
    });
);
```

Die zweite Form nimmt zwei Argumente an, nämlich eine Zuordnung von Eigenschaften und eine Zuordnung von Optionen:

```
.animate({properties}, {options})
```

Das zweite Argument ist eigentlich eine Zusammenfassung der Argumente 2 bis 4 der ersten Variante in Form einer weiteren Zuordnung, zu der jedoch noch einige erweiterte Optionen hinzukommen. Mit Zeilenumbrüchen zur Erhöhung der Lesbarkeit sieht die zweite Form wie folgt aus:

```
.animate({
    property1: 'value1',
    property2: 'value2'
}, {
    duration: 'value',
    easing: 'value',
    specialEasing: {
        property1: 'easing1',
        property2: 'easing2'
    },
    complete: function() {
        alert('The animation is finished.');
    },
    queue: true,
    step: callback
});
```

Im Folgenden verwenden wir zunächst die erste Form von `.animate()`, allerdings werden wir weiter hinten in diesem Kapitel, wenn es um aneinander gereihte Effekte geht, auf die zweite Form zurückkommen.

#### 4.4.1 Effekte manuell erstellen

Wir haben uns bereits einige vorgefertigte Effekte zum Anzeigen und Verbergen von Elementen angesehen. Zur Einführung in die Methode `.animate()` betrachten wir nun, wie wir mit dieser Schnittstelle, die mehr Handarbeit erfordert, aber auch größere Kontrolle bietet, das gleiche Resultat erzielen können wie mit dem Aufruf von `.slideToggle()`. Wie der folgende Codeausschnitt zeigt, ist es ziemlich einfach, die Zeile mit `.slideToggle()` im vorstehenden Beispiel durch eine benutzerdefinierte Animation zu ersetzen:

```
$(document).ready(function() {
    var $firstPara = $('p').eq(1);
    $firstPara.hide();
    $('a.more').click(function() {
        $firstPara.animate({height: 'toggle'}, 'slow');
        var $link = $(this);
```

```
if ($link.text() == 'read more') {  
    $link.text('read less');  
} else {  
    $link.text('read more');  
}  
return false;  
});  
});
```

**Listing 4–13**

Dies ist kein perfekter Ersatz für `.slideToggle()`. Die tatsächliche Implementierung animiert auch Außen- (Margin) und Innenrand (Padding) der Elemente.

Wie dieses Beispiel zeigt, bietet die Methode `.animate()` bequeme Kurzformen der Werte für CSS-Eigenschaften – 'show', 'hide' und 'toggle' –, die es uns erleichtern, das Verhalten von vorgefertigten Effektmethoden wie `.slideToggle()` nachzuahmen.

#### 4.4.2 Mehrere Eigenschaften gleichzeitig animieren

Mit der Methode `.animate()` können wir beliebige Kombinationen von Eigenschaften gleichzeitig animieren. Um beim Umschalten des zweiten Absatzes beispielsweise einen Falt- und Einblendeffekt zu erzielen, fügen wir zu der Eigenschaftenzuordnung von `.animate()` einfach das Eigenschaft-Wert-Paar `opacity` hinzu:

```
$(document).ready(function() {  
    var $firstPara = $('p').eq(1);  
    $firstPara.hide();  
    $('a.more').click(function() {  
        $firstPara.animate({  
            opacity: 'toggle',  
            height: 'toggle'  
        }, 'slow');  
        var $link = $(this);  
        if ($link.text() == 'read more') {  
            $link.text('read less');  
        } else {  
            $link.text('read more');  
        }  
        return false;  
    });  
});
```

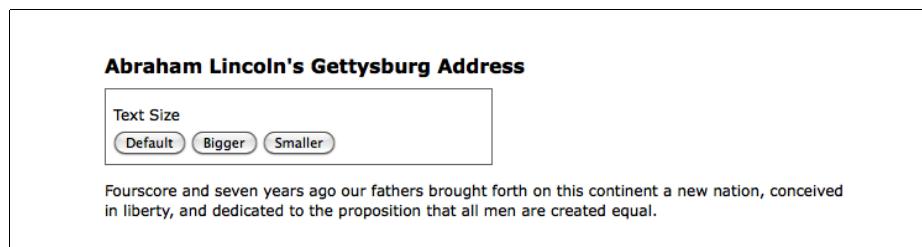
**Listing 4–14**

Außerdem stehen uns nicht nur die Formateigenschaften zur Verfügung, die in den Kurzformen der Effektmethoden verwendet werden, sondern auch numerische CSS-Eigenschaften wie `left`, `top`, `fontSize`, `margin`, `padding` und `borderWidth`. In Listing 4–5 haben wir die Textgröße der speech-Absätze geändert. Diese Vergrößerung oder Verkleinerung der Schrift können wir animieren, indem wir einfach die Methode `.css()` durch `.animate()` ersetzen:

```
$(document).ready(function() {
    var $speech = $('div.speech');
    var defaultSize = $speech.css('fontSize');
    $('#switcher button').click(function() {
        var num = parseFloat($speech.css('fontSize'));
        switch (this.id) {
            case 'switcher-large':
                num *= 1.4;
                break;
            case 'switcher-small':
                num /= 1.4;
                break;
            default:
                num = parseFloat(defaultSize);
        }
        $speech.animate({fontSize: num + 'px'}, 'slow');
    });
});
```

**Listing 4–15**

Mit den zusätzlichen Eigenschaften können wir auch weit vielschichtigere Effekte erstellen. Beispielsweise ist es möglich, ein Element von der linken Seite zur rechten zu verschieben und gleichzeitig seine Höhe um 20 Pixel und seinen Rand um 5 Pixel zu erweitern. Diese komplizierte Kombination von Eigenschaftsanimationen veranschaulichen wir am Beispiel des Kastens `<div id="switcher">`. Die folgende Abbildung zeigt, wie er vor der Animation aussieht:



Bei einem Layout mit flexibler Breite müssen wir berechnen, wie weit der Kasten verschoben werden muss, bevor er bündig am rechten Seitenrand ausgerichtet ist. Wenn wir davon ausgehen, dass die Absatzbreite 100% ist, können wir die Breite des Kastens TEXT SIZE von der Absatzbreite subtrahieren. Um diese Breiten ein-

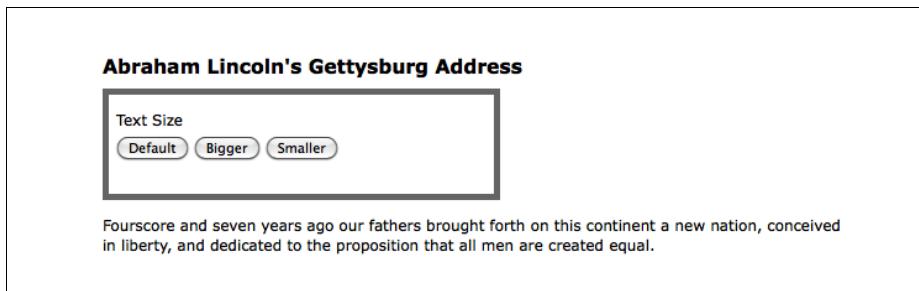
schließlich Auffüllung und Rand zu bestimmen, steht uns die jQuery-Methode `.outerWidth()` zur Verfügung. Damit berechnen wir den Wert der neuen Eigenschaft `left` für den Formatwechsler. In diesem Beispiel lösen wir die Animation durch einen Klick auf die Beschriftung TEXT SIZE oberhalb der Schaltflächen aus. Das folgende Listing zeigt den Code:

```
$(document).ready(function() {
    $('div.label').click(function() {
        var paraWidth = $('div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher.animate({
            borderWidth: '5px',
            left: paraWidth - switcherWidth,
            height: '+=20px'
        }, 'slow');
    });
});
```

**Listing 4-16**

Sehen wir uns die animierten Eigenschaften genauer an. Die Eigenschaft `borderWidth` bereitet keine Probleme, da wir einen konstanten Wert mit Einheiten angeben, wie wir es in einem Stylesheet tun würden. Dagegen ist die Eigenschaft `left` ein berechneter numerischer Wert. Bei diesen Eigenschaften ist das Suffix für die Einheit optional. Da wir es hier weglassen, wird `px` angenommen. Die Eigenschaft `height` verwendet eine Syntax, die wir noch nicht kennen. Das Präfix `=+` eines Eigenschaftswerts bezeichnet einen relativen Wert. Hier wird die Höhe also nicht auf den Wert von 20 Pixeln animiert, sondern auf einen Wert, der 20 Pixel über dem gegenwärtigen Wert liegt. Da relative Werte Sonderzeichen enthalten, müssen sie als Strings angegeben, also in Anführungszeichen geschrieben werden.

Dieser Code erhöht zwar die Höhe des `<div>`-Elements und verbreitert den Rand, doch die Position `left` bleibt unverändert, wie der folgende Screenshot zeigt:



Uns bleibt also noch die Aufgabe, die Positionsänderung des Kastens in CSS möglich zu machen.

## Positionierung in CSS

Bei der Arbeit mit `.animate()` müssen wir die Einschränkungen im Hinterkopf behalten, die CSS den Elementen auferlegt, die wir ändern möchten. Beispielsweise hat die Änderung der Eigenschaft `left` keine Auswirkung auf die ausgewählten Elemente, solange wir deren CSS-Position nicht auf `relative` oder `absolute` setzen. Die standardmäßige CSS-Position für alle Blockelemente ist `static`. Wenn wir versuchen, sie zu verschieben, ohne zuvor ihren `position`-Wert zu ändern, bleiben sie, wie diese Bezeichnung deutlich macht, statisch.

Weitere Informationen über absolute und relative Positionierung finden Sie im Artikel *Absolutely Relative* von Joe Gillespie unter [http://www.wpdtd.com/issues/78/absolutely\\_relative/](http://www.wpdtd.com/issues/78/absolutely_relative/).

Die relative Positionierung für `<div id="switcher">` könnten wir in unserem Style-sheet vornehmen:

```
#switcher {  
    position: relative;  
}
```

Stattdessen wollen wir aber unsere jQuery-Fähigkeiten einüben und diese Eigenschaft bei Bedarf mithilfe von JavaScript ändern:

```
$(document).ready(function() {  
    $('div.label').click(function() {  
        var paraWidth = $('div.speech p').outerWidth();  
        var $switcher = $(this).parent();  
        var switcherWidth = $switcher.outerWidth();  
        $switcher.css({  
            position: 'relative'  
        }).animate({  
            borderWidth: '5px',  
            left: paraWidth - switcherWidth,  
            height: '+=20px'  
        }, 'slow');  
    });  
});
```

### ***Listing 4–17***

Nachdem wir jetzt die CSS-Positionierung berücksichtigt haben, sieht das Ergebnis der Animation nach einem Klick auf TEXT SIZE folgendermaßen aus:



## 4.5 Gleichzeitige und aneinandergereihte Effekte

Wie wir gesehen haben, ist die Methode `.animate()` sehr nützlich, um Effekte zu erzielen, die gleichzeitig auf eine Menge bestimmter Elemente angewandt werden. Manchmal wollen wir unsere Effekte jedoch aneinanderreihen, sodass einer nach dem anderen wirksam wird.

### 4.5.1 Mit einem einzelnen Satz von Elementen arbeiten

Wenn wir auf einen Satz von Elementen mehrere Effekte anwenden, können wir sie einfach durch Verketten *aneinanderreihen*. Um dies zu veranschaulichen, beschäftigen wir uns erneut mit Listing 4–17, in dem wir den Kasten TEXT SIZE nach rechts verschoben und seine Höhe sowie die Breite seines Randes erweitert haben. Diesmal aber führen wir die drei Effekte nacheinander aus, indem wir sie einfach jeweils in ihrer eigenen `.animate()`-Methode platzieren und die drei Methoden verketten:

```
$(document).ready(function() {
    $('#div.label').click(function() {
        var paraWidth = $('#div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher
            .css({position: 'relative'})
            .animate({left: paraWidth - switcherWidth}, 'slow')
            .animate({height: '+=20px'}, 'slow')
            .animate({borderWidth: '5px'}, 'slow');
    });
});
```

**Listing 4–18**

Wie Sie wissen, können wir bei der Verkettung alle drei `.animate()`-Methoden in einer Zeile schreiben, aber hier haben wir sie jeweils in eine eigene Zeile geschrieben und sie eingerückt, um die Lesbarkeit zu verbessern.

Wir können nicht nur `.animate()`, sondern alle jQuery-Effektmethoden aneinanderreihen, indem wir sie verketten. Beispielsweise können wir die Effekte für `<div id="switcher">` in folgender Reihenfolge anordnen:

1. Verringern der Deckkraft auf 0,5 mit `.fadeTo()`.
2. Verschieben nach rechts mit `.animate()`.
3. Wiederherstellen der vollen Deckkraft mit `.fadeTo()`.
4. Verbergen mit `.slideUp()`.
5. Erneut anzeigen mit `.slideDown()`.

Dazu müssen wir nichts anderes tun, als die Effekte in dieser Reihenfolge im Code zu verketten:

```
$(document).ready(function() {
    $('#div.label').click(function() {
        var paraWidth = $('div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher
            .css({position: 'relative'})
            .fadeTo('fast', 0.5)
            .animate({left: paraWidth - switcherWidth}, 'slow')
            .fadeTo('slow', 1.0)
            .slideUp('slow')
            .slideDown('slow');
    });
});
```

**Listing 4-19**

### Die Aneinanderreichung umgehen

Was aber ist zu tun, wenn wir das `<div>`-Element in dem Zeitraum nach rechts verschieben möchten, in dem seine Deckkraft auf die Hälfte verringert wird? Wenn die beiden Animationen dieselbe Geschwindigkeit hätten, könnten wir sie einfach in einer einzigen `.animate()`-Methode kombinieren. In diesem Beispiel hat die Deckkraftänderung jedoch die Geschwindigkeit 'fast', die Verschiebung nach rechts dagegen die Geschwindigkeit 'slow'. Im folgenden Codeausschnitt nutzen wir für dieses Problem die zweite Form der Methode `.animate()`:

```
$(document).ready(function() {
    $('#div.label').click(function() {
        var paraWidth = $('div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher
```

```
.css({position: 'relative'})  
.fadeTo('fast', 0.5)  
.animate({  
    left: paraWidth - switcherWidth  
}, {  
    duration: 'slow',  
    queue: false  
})  
.fadeTo('slow', 1.0)  
.slideUp('slow')  
.slideDown('slow');  
});  
});
```

**Listing 4–20**

Das zweite Argument, die Optionszuordnung, enthält die Option `queue`. Wenn wir sie auf `false` setzen, beginnt die Animation gleichzeitig mit der vorhergehenden.

### Manuelle Aneinanderreihung

Eine letzte Anmerkung zu aneinandergereihten Effekten für einen einzigen Satz von Elementen gibt es noch: Andere Methoden, die nicht für Effekte zuständig sind, z.B. `.css()`, werden nicht automatisch aneinandergereiht. Betrachten wir als Beispiel den Fall, dass wir die Hintergrundfarbe von `<div id="switcher">` nach `.slideUp()`, aber vor `.slideDown()` in Rot ändern wollen.

Dazu können wir folgenden Code verwenden:

```
// Unfinished code  
$(document).ready(function() {  
    $('#div.label').click(function() {  
        var paraWidth = $('div.speech p').outerWidth();  
        var $switcher = $(this).parent();  
        var switcherWidth = $switcher.outerWidth();  
        $switcher  
            .css({position: 'relative'})  
            .fadeTo('fast', 0.5)  
            .animate({  
                left: paraWidth - switcherWidth  
            }, {  
                duration: 'slow',  
                queue: false  
            })  
            .fadeTo('slow', 1.0)  
            .slideUp('slow')  
            .css({backgroundColor: '#f00'})  
            .slideDown('slow');  
    });  
});
```

**Listing 4–21**

Obwohl der Code für die Änderung der Hintergrundfarbe an der richtigen Stelle der Kette steht, erfolgt die Umfärbung sofort beim Klicken.

Eine Möglichkeit, um Nicht-Effektmethoden in die Reihe einzugliedern, bietet die Methode `.queue()`:

```
$(document).ready(function() {
    $('div.label').click(function() {
        var paraWidth = $('div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher
            .css({position: 'relative'})
            .fadeTo('fast', 0.5)
            .animate({
                left: paraWidth - switcherWidth
            }, {
                duration: 'slow',
                queue: false
            })
            .fadeTo('slow', 1.0)
            .slideUp('slow')
            .queue(function(next) {
                $switcher.css({backgroundColor: '#f00'});
                next();
            })
            .slideDown('slow');
    });
});
```

**Listing 4-22**

Wenn `.queue()` wie hier eine Callback-Funktion übergeben wird, fügt sie die Funktion der Effektreihe hinzu, sodass sie auf die ausgewählten Elemente angewendet wird. Innerhalb der Funktion setzen wir die Hintergrundfarbe auf Rot und rufen dann `next()` auf, eine Funktion, die als Parameter an unseren Callback übergeben wird. Dank der Verwendung des Funktionsaufrufs `next()` kann die Animationsreihe dort fortgesetzt werden, wo sie unterbrochen wurde, und die Kette mit der folgenden Zeile – `.slideDown('slow')` – abschließen. Ohne den Aufruf von `next()` wäre die Animation abgebrochen worden.

Weitere Informationen zu `.queue()` sowie Anwendungsbeispiele finden Sie unter <http://api.jquery.com/effects/>.

Eine weitere Methode zum Einschalten von Nicht-Effektmethoden in solche Aneinanderreihungen sehen wir uns an, wenn wir Effekte für mehrere Sätze von Elementen betrachten.

#### 4.5.2 Mit mehreren Sätzen von Elementen arbeiten

Wenn wir Effekte auf mehrere Sätze anwenden, treten sie im Gegensatz zu den Effekten für einen einzigen Satz praktisch zur selben Zeit auf. Um diese simultanen Effekte in Aktion zu sehen, falten wir einen Absatz auseinander, während wir einen anderen zusammenfalten. Dabei arbeiten wir mit den Absätzen 3 und 4 unseres Beispieldokuments:

```
<p>Four score and seven years ago our fathers brought forth  
on this continent a new nation, conceived in liberty,  
and dedicated to the proposition that all men are  
created equal.</p>
```

```
<p>Now we are engaged in a great civil war, testing whether  
that nation, or any nation so conceived and so  
dedicated, can long endure. We are met on a great  
battlefield of that war. We have come to dedicate a  
portion of that field as a final resting-place for those  
who here gave their lives that the nation might live. It  
is altogether fitting and proper that we should do this.  
But, in a larger sense, we cannot dedicate, we cannot  
consecrate, we cannot hallow, this ground.</p>
```

[read more](#)

```
<p>The brave men, living and dead, who struggled here have  
consecrated it, far above our poor power to add or  
detract. The world will little note, nor long remember,  
what we say here, but it can never forget what they did  
here. It is for us the living, rather, to be dedicated  
here to the unfinished work which they who fought here  
have thus far so nobly advanced.</p>
```

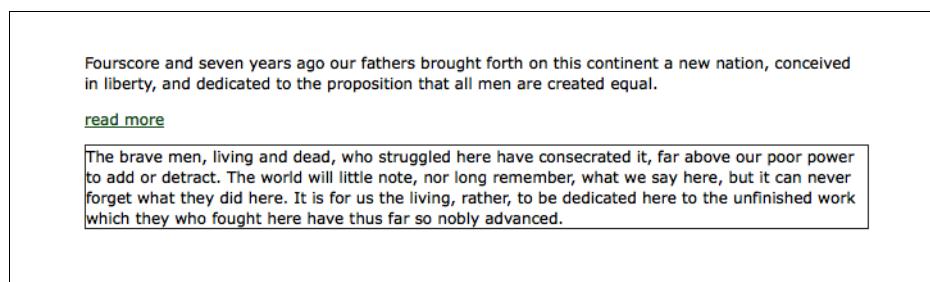
```
<p>It is rather for us to be here dedicated to the great  
task remaining before us&mdash;that from these honored  
dead we take increased devotion to that cause for which  
they gave the last full measure of devotion&mdash;that  
we here highly resolve that these dead shall not have  
died in vain&mdash;that this nation, under God, shall  
have a new birth of freedom and that government of the  
people, by the people, for the people, shall not perish  
from the earth.</p>
```

Um besser erkennen zu können, was bei diesem Effekt geschieht, versehen wir den dritten Absatz mit einem Rand von einem Pixel Breite und den vierten Absatz mit einem grauen Hintergrund. Außerdem verbergen wir den vierten Absatz, wenn das DOM bereitsteht. Das alles können Sie im folgenden Codeausschnitt sehen:

```
$(document).ready(function() {
    $('p').eq(2).css('border', '1px solid #333');
    $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

**Listing 4-23**

Unser Beispieldokument zeigt jetzt den einleitenden Absatz gefolgt von dem Link READ MORE und dem Absatz mit dem Rahmen:



Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

[read more](#)

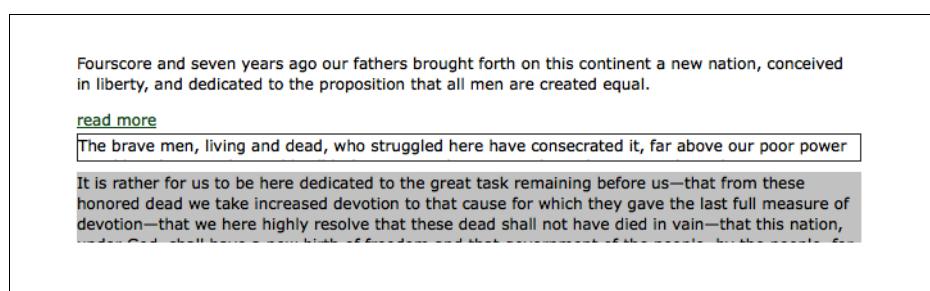
The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced.

Schließlich fügen wir dem dritten Absatz noch die Methode `.click()` hinzu, damit er, wenn jemand darauf klickt, zusammengefaltet wird (und damit verschwindet), während der vierte Absatz auseinandergefaltet (und damit sichtbar) wird:

```
$(document).ready(function() {
    $('p').eq(2)
        .css('border', '1px solid #333')
        .click(function() {
            $(this).slideUp('slow').next().slideDown('slow');
        });
    $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

**Listing 4-24**

Ein Screenshot dieser beiden Effekte, aufgenommen in der Mitte des Vorgangs, bestätigt, dass die Effekte tatsächlich gleichzeitig ablaufen:



Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

[read more](#)

The brave men, living and dead, who struggled here have consecrated it, far above our poor power

It is rather for us to be here dedicated to the great task remaining before us—that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion—that we here highly resolve that these dead shall not have died in vain—that this nation,

Der dritte Absatz, der zu Anfang sichtbar ist, wird gerade zusammengefaltet, während der ursprünglich verborgene vierte Absatz gerade auseinandergefaltet wird.

### Callbacks

Um die Aneinanderreihung von Effekten für verschiedene Elemente zu ermöglichen, bietet jQuery für jede Effektmethode eine *Callback-Funktion*. Wie wir bereits bei Ereignishandlern und bei der Methode `.queue()` gesehen haben, sind Callbacks nichts anderes als Funktionen, die als Argumente an Methoden übergeben werden. Bei Effekten bilden sie jeweils das letzte Argument der Methode.

Wenn wir einen Callback verwenden, um die beiden Falteffekte aneinanderzuriehen, können wir den vierten Absatz auseinanderfalten lassen, bevor der dritte zusammengefaltet wird. Versuchen wir als Erstes, den Aufruf von `.slideUp()` in den abschließenden Callback der Methode `.slideDown()` zu verschieben:

```
$(document).ready(function() {
    $('p').eq(2)
        .css('border', '1px solid #333')
        .click(function() {
            $(this).next().slideDown('slow', function() {
                $(this).slideUp('slow');
            });
        });
    $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

**Listing 4–25**

Wir müssen jedoch aufpassen, was hier zusammengefaltet wird. Der Kontext der Funktion – das Schlüsselwort `this` – ist hier ein anderer, da sich der Callback innerhalb der Methode `.slideDown()` befindet. Hier ist `$(this)` nicht mehr der dritte Absatz wie unmittelbar innerhalb des `click`-Handlers. Da die Methode `.slideDown()` als Ergebnis von `$(this).next()` aufgerufen wird, sieht der Callback innerhalb dieser Methode jetzt `$(this)` als das nächste Geschwisterelement an, also den vierten Absatz. Wenn wir `$(this).slideUp('slow')` in dem Callback platzieren, wie wir es in Listing 4–25 getan haben, verbergen wir dadurch den Absatz, den wir gerade erst sichtbar gemacht haben.

Eine einfache Möglichkeit, um den Verweis `$(this)` unverändert zu belassen, besteht darin, ihn gleich im `click`-Handler in einer Variablen zu speichern, z.B. in `var $clickedItem = $(this)`.

Jetzt verweist \$clickedItem sowohl innerhalb als auch außerhalb des Effektmethoden-Callbacks auf den dritten Absatz. Mit dieser neuen Variablen sieht unser Code wie folgt aus:

```
$(document).ready(function() {
    $('p').eq(2)
        .css('border', '1px solid #333')
        .click(function() {
            var $clickedItem = $(this);
            $clickedItem.next().slideDown('slow', function() {
                $clickedItem.slideUp('slow');
            });
        });
    $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

**Listing 4–26**

Voraussetzung für die Verwendung von \$clickedItem innerhalb des .slideDown()-Callbacks ist die Eigenschaft von *Funktionseinschlüssen (Closures)*. Dieses wichtige, aber schwierig zu beherrschende Thema besprechen wir in Anhang A.

Eine Momentaufnahme mitten im Ablauf der Effekte zeigt jetzt sowohl den dritten als auch den vierten Absatz im sichtbaren Zustand. Der vierte ist gerade auseinandergefaltet worden, während der dritte beginnt, sich zusammenzufalten:

Fourscore and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and dedicated to the proposition that all men are created equal.

[read more](#)

The brave men, living and dead, who struggled here have consecrated it, far above our poor power to add or detract. The world will little note, nor long remember, what we say here, but it can never forget what they did here. It is for us the living, rather, to be dedicated here to the unfinished work which they who fought here have thus far so nobly advanced.

It is rather for us to be here dedicated to the great task remaining before us—that from these honored dead we take increased devotion to that cause for which they gave the last full measure of devotion—that we here highly resolve that these dead shall not have died in vain—that this nation, under God, shall have a new birth of freedom and that government of the people, by the people, for

Da wir uns jetzt mit Callbacks auskennen, können wir zum Code aus Listing 4–22 zurückkehren, in dem wir am Ende einer Reihe von Effekten eine Änderung der Hintergrundfarbe eingefügt haben. Anstatt wie dort die Methode .queue() zu verwenden, können wir einfach wie folgt eine Callback-Funktion einsetzen:

```
$(document).ready(function() {
    $('div.label').click(function() {
        var paraWidth = $('div.speech p').outerWidth();
        var $switcher = $(this).parent();
        var switcherWidth = $switcher.outerWidth();
        $switcher
            .css({position: 'relative'})
            .fadeTo('fast', 0.5)
            .animate({
                left: paraWidth - switcherWidth
            }, {
                duration: 'slow',
                queue: false
            })
            .fadeTo('slow', 1.0)
            .slideUp('slow', function() {
                $switcher.css({backgroundColor: '#f00'});
            })
            .slideDown('slow');
    });
});
```

**Listing 4-27**

Wiederum wird die Hintergrundfarbe von <div id="switcher"> in Rot geändert, nachdem der Bereich zusammengefaltet wurde, aber bevor er wieder auseinandergefaltet wird. Wenn Sie statt .queue() den abschließenden Callback eines Effekts aufrufen, brauchen Sie nicht next() innerhalb des Callbacks aufzurufen.

#### 4.5.3 Kurz und bündig

Angesichts all der Variationsmöglichkeiten bei der Anwendung von Effekten kann es schwierig sein, sich zu merken, ob die Effekte nun gleichzeitig oder nacheinander ablaufen. Die folgende kleine Übersicht mag zur Orientierung hilfreich sein:

1. Effekte für einen einzelnen Satz von Elementen laufen wie folgt ab:
  - **Gleichzeitig**, wenn sie in Form mehrerer Eigenschaften in derselben .animate()-Methode angewandt werden.
  - **Aneinandergereiht**, wenn sie in einer Kette von Methoden angewandt werden, es sei denn, die Option queue ist auf false gesetzt.
2. Effekte für mehrere Sätze von Elementen laufen wie folgt ab:
  - **Gleichzeitig** nach Voreinstellung.
  - **Aneinandergereiht**, wenn sie im Callback eines anderen Effekts oder im Callback der Methode .queue() angewandt werden.

## 4.6 Zusammenfassung

Mit den in diesem Kapitel vorgestellten Effektmethoden können wir Inline-Formatattribute mit JavaScript bearbeiten, vorgefertigte jQuery-Effekte auf Elemente anwenden und eigene benutzerdefinierte Animationen erstellen. Insbesondere haben wir gelernt, wie wir die Schriftgröße inkrementell mit .css und .animate() vergrößern und verringern, wie wir Seitenelemente durch die Bearbeitung verschiedener Attribute nach und nach verbergen und anzeigen und wie wir auf verschiedene Weise Elemente gleichzeitig und nacheinander animieren.

In den ersten vier Kapiteln dieses Buches haben wir in den Beispielen nur Elemente bearbeitet, die im HTML-Markup der Seite hart codiert waren. In Kapitel 5 werden wir dagegen Möglichkeiten kennenlernen, das DOM direkt zu bearbeiten. Unter anderem werden wir jQuery verwenden, um neue Elemente zu erstellen und sie an beliebiger Stelle ins DOM einzufügen.

### 4.6.1 Literatur

Das Thema Animation wird ausführlicher in Kapitel 11 behandelt. Eine vollständige Liste der Effekt- und Formatierungsmethoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## 4.7 Übungsaufgaben

Um die folgenden Übungen durchführen zu können, benötigen Sie die Datei index.html für dieses Kapitel sowie den fertigen JavaScript-Code aus complete.js. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Ändern Sie das Stylesheet, sodass der Seiteninhalt zu Anfang verborgen ist. Lassen Sie die Inhalte nach dem Laden der Seite langsam einblenden.
2. Sorgen Sie dafür, dass die Absätze mit einem gelben Hintergrund dargestellt werden, wenn der Benutzer mit dem Mauszeiger darüberfährt.
3. Sorgen Sie dafür, dass der Titel (das <h2>-Element) beim Anklicken gleichzeitig auf eine Deckkraft von 25% abgesenkt wird und einen linken Rand von 20 Pixel Breite erhält. Nach dem Abschluss dieser Animation soll die Deckkraft des Textes der Rede auf 50% abgesenkt werden.
4. Schwierig: Reagieren Sie auf die Betätigung der Pfeiltasten, indem Sie den Kasten des Formatwechslers um 20 Pixel in die entsprechende Richtung verschieben. Die Codes für die Pfeiltasten sind 37 (nach links), 28 (nach oben), 39 (nach rechts) und 40 (nach unten).

## 5 DOM-Bearbeitung

Was wir im Web sehen, kommt durch die Zusammenarbeit zwischen Webserver und Webbrowser zustande. Traditionell war es die Aufgabe des Servers, ein fertiges HTML-Dokument für die Nutzung durch den Browser bereitzustellen. Die bisher behandelten Techniken haben diese Aufteilung ein wenig verschoben, da wir das Erscheinungsbild des HTML-Dokuments mithilfe von CSS im laufenden Betrieb geändert haben. Um unsere JavaScript-Muskeln aber richtig spielen zu lassen, müssen wir lernen, wie wir das Dokument selbst ändern können.

Mit jQuery können wir das Dokument auf einfache Weise ändern, indem wir die vom DOM (Document Object Model) bereitgestellte Schnittstelle nutzen. Dadurch können wir Elemente und Text auf einer Webseite erstellen, wann immer wir es brauchen. Außerdem können wir all diese Dinge auch im laufenden Betrieb verschwinden lassen oder umwandeln, indem wir Attribute hinzufügen, entfernen oder ändern.

### 5.1 Attribute bearbeiten

In den ersten vier Kapiteln dieses Buches haben wir die Methoden `.addClass()` und `.removeClass()` verwendet, um zu demonstrieren, wie wir das Erscheinungsbild von Elementen auf einer Seite ändern können. Letzten Endes machen diese beiden Methoden nichts anderes, als das Attribut `class` (im DOM-Jargon »die Eigenschaft `className`«) des Elements zu bearbeiten. Die Methode `.addClass()` erstellt dieses Attribut oder fügt es hinzu, die Methode `.removeClass()` löscht oder kürzt es. Wenn Sie das mit der Methode `.toggleClass()` kombinieren, die zwischen dem Hinzufügen und Entfernen einer Klasse wechselt, haben Sie eine effiziente und robuste Möglichkeit zur Bearbeitung von Klassen.

Aber das Attribut `class` ist nur eines von vielen, auf die wir zugreifen oder die wir ändern möchten. Beispielsweise gibt es noch `id`, `rel` und `href`. Um sie zu bearbeiten, bietet jQuery die Methoden `.attr()` und `.removeAttr()`. Wir könnten `.attr()` und `.removeAttr()` sogar zur Bearbeitung von `class` einsetzen, aber die

spezialisierten Klassen `.addClass()` und `.removeClass()` eignen sich dafür besser, da sie Fälle, in denen mehrere Klassen zu einem einzigen Element hinzufügt werden (wie in `<div class="first second">`), korrekt handhaben.

### 5.1.1 Nicht-Klassenattribute

Manche Attribute lassen sich ohne die Hilfe von jQuery nicht so einfach bearbeiten. Außerdem erlaubt uns jQuery, mehrere Attribute auf einmal zu ändern, ähnlich wie wir in Kapitel 4, »Formatierung und Animation«, mit `.css()` an CSS-Eigenschaften gearbeitet haben.

Beispielsweise können wir die Attribute `id`, `rel` und `title` für Links ganz einfach alle auf einmal festlegen. Als Erstes sehen wir uns dazu den folgenden HTML-Beispielcode an:

```
<h1 id="f-title">Flatland: A Romance of Many Dimensions</h1>
<div id="f-author">by Edwin A. Abbott</div>
<h2>Part 1, Section 3</h2>
<h3 id="f-subtitle">
    Concerning the Inhabitants of Flatland
</h3>
<div id="excerpt">an excerpt</div>
<div class="chapter">
    <p class="square">Our Professional Men and Gentlemen are
        Squares (to which class I myself belong) and Five-Sided
        Figures or <a href="http://en.wikipedia.org/wiki/Pentagon">Pentagons
        </a>.
    </p>
    <p class="nobility hexagon">Next above these come the
        Nobility, of whom there are several degrees, beginning at
        Six-Sided Figures, or <a href="http://en.wikipedia.org/wiki/Hexagon">Hexagons</a>,
        and from thence rising in the number of their sides till
        they receive the honourable title of <a href="http://en.wikipedia.org/wiki/Polygon">Polyagonal</a>,
        or many-Sided. Finally when the number of the sides
        becomes so numerous, and the sides themselves so small,
        that the figure cannot be distinguished from a <a href="http://en.wikipedia.org/wiki/Circle">circle</a>, he
        is included in the Circular or Priestly order; and this is
        the highest class of all.
    </p>
    <p><span class="pull-quote">It is a Law
        of Nature <span class="drop">with us</span> that a male child shall
        have<strong>one more side</strong> than his father</span>, so
        that each generation shall rise (as a rule) one step in
        the scale of development and nobility. Thus the son of a
```

## 5.1 Attribute bearbeiten

---

```
        Square is a Pentagon; the son of a Pentagon, a Hexagon;  
        and so on.  
    </p>  
<!-- . . . code continues . . . -->  
</div>
```

Wir können nun durch die einzelnen Links innerhalb von `<div class="chapter">` iterieren und die Attribute nacheinander auf sie anwenden. Wenn wir ein Attribut für alle Links auf denselben Wert setzen müssen, können wir das mit einer einzigen Codezeile im Handler `$(document).ready()` erledigen:

```
$(document).ready(function() {  
    $('div.chapter a').attr({rel: 'external'});  
});
```

### ***Listing 5–1***

Wie die Methode `.css()` kann auch `.attr()` ein Paar von Parametern annehmen, von denen der erste Parameter einen Attributnamen angibt und der zweite den neuen Wert dieses Attributs. Gewöhnlich jedoch stellen wir eine *Zuordnung (Map)* von Schlüssel-Wert-Paaren bereit wie in Listing 5–1. Mit dieser Syntax können wir unser Beispiel ohne Schwierigkeiten erweitern, um mehrere Attribute gleichzeitig zu bearbeiten:

```
$(document).ready(function() {  
    $('div.chapter a').attr({  
        rel: 'external',  
        title: 'Learn more at Wikipedia'  
    });  
});
```

### ***Listing 5–2***

## **Wert-Callbacks**

Die einfache Technik, `.attr()` eine Zuordnung von Konstanten zu übergeben, reicht aus, wenn die Attribute in allen ausgewählten Elementen denselben Wert haben sollen. Häufig jedoch müssen die Attribute, die wir hinzufügen oder ändern, jedes Mal einen anderen Wert aufweisen. Ein häufig vorkommendes Beispiel ist der `id`-Wert, der innerhalb eines Dokuments eindeutig sein muss, damit sich der JavaScript-Code vorhersehbar verhält. Um für jeden Link eine eindeutige `id` festzulegen, können wir ein anderes Merkmal von jQuery-Methoden wie `.css()` und `.each()` nutzen: den *Wert-Callback*.

Ein Wert-Callback ist einfach eine Funktion, die anstelle des Werts als Argument übergeben wird. Diese Funktion wird für jedes Element in der ausgewählten Menge einmal aufgerufen, und die Daten, die diese Funktion zurückgibt, werden dann als neuer Wert für das Attribut verwendet. Beispielsweise können wir mit dieser Technik für jedes Element einen anderen `id`-Wert erzeugen:

```
$(document).ready(function() {
    $('div.chapter a').attr({
        rel: 'external',
        title: 'Learn more at Wikipedia',
        id: function(index, oldValue) {
            return 'wikilink-' + index;
        }
    });
});
```

**Listing 5–3**

Jedes Mal, wenn der Wert-Callback ausgelöst wird, wird ihm ein Parameterpaar übergeben. Der erste Parameter ist eine Ganzzahl (ein Integerwert), die die laufende Nummer der Iteration angibt. Wir verwenden sie hier, um dem ersten Link den id-Wert `wikilink-0` zu geben, dem zweiten `wikilink-1` usw. Der zweite Parameter, der in Listing 5–3 nicht verwendet wird, enthält den Wert des Attributs vor der Änderung.

Wir benutzen das Attribut `title`, um die Leser dazu einzuladen, in Wikipedia mehr über den verlinkten Begriff nachzulesen. In dem HTML-Code, den wir bisher verwendet haben, zeigen alle Links auf Wikipedia. Um jedoch auch andere Arten von Links zu berücksichtigen, müssen wir die Selektorausdrücke etwas präziser gestalten:

```
$(document).ready(function() {
    $('div.chapter a[href*="wikipedia"]').attr({
        rel: 'external',
        title: 'Learn more at Wikipedia',
        id: function(index, oldValue) {
            return 'wikilink-' + index;
        }
    });
});
```

**Listing 5–4**

Um unsere Einführung der Methode `.attr()` abzuschließen, erweitern wir das `title`-Attribut dieser Links, sodass es mehr über das Linkziel verrät. Auch hier ist ein Wert-Callback das richtige Instrument für diese Aufgabe:

```
$(document).ready(function() {
    $('div.chapter a[href*="wikipedia"]').attr({
        rel: 'external',
        title: function() {
            return 'Learn more about ' + $(this).text()
                + ' at Wikipedia.';
        },
        id: function(index, oldValue) {
            return 'wikilink-' + index;
        }
    });
});
```

## 5.1 Attribute bearbeiten

113

```
});  
});
```

**Listing 5-5**

Diesmal haben wir den *Kontext* der Wert-Callbacks genutzt. Wie bei Ereignishandlern zeigt das Schlüsselwort `this` auf das DOM-Element, das wir bearbeiten, wenn der Callback aufgerufen wird. Hier verpacken wir das Element in ein jQuery-Objekt, sodass wir die (in Kapitel 4 eingeführte) Methode `.text()` verwenden können, um den Textinhalt des Links abzurufen. Dadurch werden unsere Linktitel erfreulich genau:

**Flatland: A Romance of Many Dimensions**  
by Edwin A. Abbott

**Part 1, Section 3**

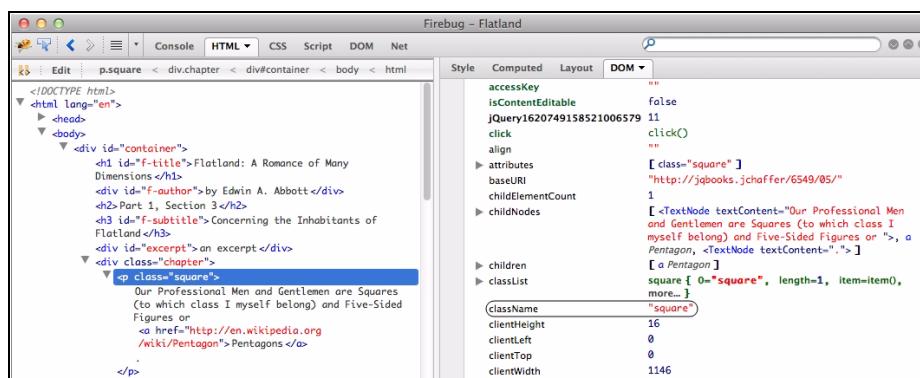
**Concerning the Inhabitants of Flatland**  
*an excerpt*

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or [Pentagons](#).

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or [Hexagons](#), and from thence rising in the number of their sides till they receive the honourable title of [Polvaonal](#). or many-Sided. Finally when the number

### 5.1.2 Eigenschaften von DOM-Elementen

Wie zuvor schon angedeutet, gibt es einen feinen Unterschied zwischen HTML-Attributen und DOM-Eigenschaften. Attribute sind die Werte, die in Anführungszeichen im HTML-Quelltext der Seite stehen, während es sich bei Eigenschaften um die Werte handelt, auf die JavaScript zugreift. Attribute und Eigenschaften können wir gut in einem Entwicklerwerkzeug wie Firebug beobachten, wie der folgende Screenshot zeigt:



Im Firebug-Inspektor sehen wir, dass das hervorgehobene <p>-Element über das Attribut class mit dem Wert square verfügt. Im rechten Bereich können wir erkennen, dass dieses Element die entsprechende Eigenschaft className mit dem Wert square besitzt. Dies ist eine der wenigen Situationen, in denen ein Attribut und die zugehörige Eigenschaft unterschiedliche Namen aufweisen.

In den meisten Fällen sind Attribute und Eigenschaften funktional austauschbar, wobei sich jQuery für uns um Namensabweichungen kümmert. Manchmal müssen wir die Unterschiede beachten. Vor allem die Datentypen können voneinander abweichen. So hat beispielsweise das Attribut checked einen Stringwert, die Eigenschaft checked aber einen booleschen Wert. Bei solchen *booleschen Attributen* ist es am besten, anstelle des *Attributs* die *Eigenschaft* zu testen und festzulegen, um ein einheitliches Verhalten in verschiedenen Browsern sicherzustellen.

Mit der Methode .prop() können wir Eigenschaften in jQuery wie folgt abrufen und festlegen:

```
// Ruft den aktuellen Wert der Eigenschaft checked ab  
var currentlyChecked = $('.my-checkbox').prop('checked');  
  
// Legt einen neuen Wert für die Eigenschaft checked fest  
$('.my-checkbox').prop('checked', false);
```

Die Methode .prop() weist dieselben Merkmale auf wie .attr() und akzeptiert z.B. ebenfalls eine Zuordnung mehrerer gleichzeitig festzulegender Werte sowie Wert-Callback-Funktionen.

## 5.2 Bearbeitung des DOM-Baums

Die Methoden .attr() und .prop() sind leistungsfähige Werkzeuge, mit denen wir das Dokument selbst bearbeiten können. Wir kennen aber immer noch keine Möglichkeit, um die Struktur des Dokuments zu ändern. Um tatsächlich den DOM-Baum zu bearbeiten, müssen wir mehr über die Funktion lernen, die das Herz der Bibliothek jQuery bildet.

### 5.2.1 Neues zur Funktion \$()

Von Anfang an haben wir in diesem Buch die Funktion \$() verwendet, um auf Elemente in einem Dokument zuzugreifen. Wie wir gesehen haben, dient diese Funktion als Factory und produziert neue jQuery-Objekte, die auf mittels CSS-Selektoren beschriebene Elemente zeigen.

Doch das ist nicht alles, was die Funktion \$() kann. Tatsächlich ist sie so leistungsfähig, dass Sie damit nicht nur das optische Erscheinungsbild einer Seite ändern können, sondern auch deren Inhalt. Allein dadurch, dass wir der Funktion ein wenig HTML-Code übergeben, können wir eine völlig neue DOM-Struktur aus dem Hut zaubern.

### Allgemeine Zugänglichkeit

Bedenken Sie auch hier die Gefahren, die dabei lauern, wenn wir Funktionen, grafische Gestaltungen und Textinformationen nur für diejenigen verfügbar machen, deren Webbrowser JavaScript nutzen können (und in denen es aktiviert ist). Wichtige Informationen sollten für alle zugänglich sein, nicht nur für die Personen, die zufällig die richtige Software verwenden.

#### 5.2.2 Neue Elemente erstellen

Ein vor allem auf FAQ-Seiten häufig anzutreffendes Merkmal ist der Link »zurück an den Anfang« hinter jedem Frage-und-Antwort-Paar. Man kann durchaus der Meinung sein, dass diese Links keinerlei semantischen Zweck erfüllen und daher reinen Gewissens als Bonus für einen Teil der Besucher in den JavaScript-Code ausgelagert werden können. In unserem Beispiel fügen wir einen solchen Link (BACK TO TOP) hinter jedem Absatz ein und stellen auch den Anker bereit, zu dem diese Links führen. Als Erstes erzeugen wir die neuen Elemente:

```
// Unfinished code
$(document).ready(function() {
    $('<a href="#top">back to top</a>');
    $('<a id="top"></a>');
});
```

#### ***Listing 5–6***

In der ersten Codezeile haben wir einen BACK TO TOP-Link erstellt, in der zweiten den Zielanker für diesen Link. Allerdings erscheinen diese Links noch nicht auf der Seite:

Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.

Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order; and this is the highest class of all.

*It is a Law of Nature with us that a male child shall have one more side than his father, so that each generation shall rise (as it were) one*

Die beiden Codezeilen erstellen zwar tatsächlich diese Elemente, fügen sie aber nicht zu der Seite hinzu. Wir müssen dem Browser erst mitteilen, wo er sie platzieren soll. Dazu können wir eine der vielen *Einfügemethoden* von jQuery verwenden.

### 5.2.3 Neue Elemente einfügen

Um Elemente in ein Dokument einzufügen, stehen in der Bibliothek jQuery mehrere Methoden zur Verfügung, die jeweils die Beziehung zwischen dem neuen und dem vorhandenen Inhalt vorgeben. Da wir die BACK TO TOP-Links in unserem Beispiel hinter den einzelnen Absätzen einfügen möchten, verwenden wir die Methode mit dem passenden Namen `.insertAfter()`:

```
// Unfinished code
$(document).ready(function() {
    $('<a href="#top">back to top</a>').insertAfter('div.chapter p');
    $('<a id="top"></a>');
});
```

#### ***Listing 5–7***

Nachdem wir die Links jetzt auf der Seite (und im DOM) hinter den einzelnen Absätzen in `<div class="chapter">` eingefügt haben, erscheinen sie tatsächlich, wie der folgende Screenshot zeigt:

```
Our Professional Men and Gentlemen are Squares (to which class I myself belong) and Five-Sided Figures or Pentagons.
back to top
Next above these come the Nobility, of whom there are several degrees, beginning at Six-Sided Figures, or Hexagons, and from thence rising in the number of their sides till they receive the honourable title of Polygonal, or many-Sided. Finally when the number of the sides becomes so numerous, and the sides themselves so small, that the figure cannot be distinguished from a circle, he is included in the Circular or Priestly order; and this is the highest class of all.
back to top
It is a Law of Nature with us that a male child shall have one more side than his father so that each generation shall rise (as a ruler) one
```

Beachten Sie, dass die neuen Links jeweils in einer eigenen Zeile stehen und nicht in dem vorausgehenden Absatz. Das liegt daran, dass die Methode `.insertAfter()` und ihr Gegenstück `.insertBefore()` Inhalte außerhalb des angegebenen Elements platzieren.

Leider funktionieren die Links noch nicht, denn wir müssen immer noch den Anker mit `id="top"` einfügen. Dazu verwenden wir eine der Methoden, die Elemente innerhalb von anderen Elementen platzieren:

```
$(document).ready(function() {
    $('<a href="#top">back to top</a>').insertAfter('div.chapter p');
    $('<a id="top"></a>').prependTo('body');
});
```

#### ***Listing 5–8***

Dieser Code fügt den Anker unmittelbar am Anfang des `<body>`-Elements ein, also am Anfang der Seite. Dank der Methoden `.insertAfter()` für die Links und `.prependTo()` für den Anker enthält die Seite jetzt einen funktionierenden Satz von BACK TO TOP-Links.

Wenn wir noch die Methode `.appendTo()` hinzufügen, haben wir alle Möglichkeiten, um neue Elemente vor und hinter anderen Elementen einzufügen:

1. `.insertBefore()` fügt Inhalte außerhalb und vor bestehenden Elementen ein.
2. `.prependTo()` fügt Inhalte innerhalb und vor bestehenden Elementen ein.
3. `.appendTo()` fügt Inhalte innerhalb und hinter bestehenden Elementen ein.
4. `.insertAfter()` fügt Inhalte außerhalb und hinter bestehenden Elementen ein.

#### 5.2.4 Elemente verschieben

Beim Hinzufügen der BACK TO TOP-Links haben wir neue Elemente erstellt und auf der Seite eingefügt. Es ist aber auch möglich, Elemente an einer Stelle der Seite zu entnehmen und an einer anderen Stelle einzufügen. Eine praktische Anwendung dafür ist die dynamische Platzierung und Formatierung von Fußnoten. In dem ursprünglichen Flatland-Text, den wir im folgenden Beispiel verwenden, kommt bereits eine Fußnote vor, aber zur Veranschaulichung weisen wir auch andere Teile dieses Textes als Fußnoten aus.

```
<p>How admirable is the Law of Compensation!
<span class="footnote">
    And how perfect a proof of the natural
    fitness and, I may almost say, the divine origin of the
    aristocratic constitution of the States of Flatland!
</span>
By a judicious use of this Law of Nature, the Polygons and
Circles are almost always able to stifle sedition in its
very cradle, taking advantage of the irrepressible and
boundless hopefulness of the human mind.&hellip;
</p>
```

Unser HTML-Dokument enthält drei Fußnoten, wovon dieser Absatz ein Beispiel zeigt. Der Fußnotentext steht im Absatztext, abgesetzt durch `<span class="footnote"></span>`. Durch diese Art des HTML-Markups bleibt der Kontext der Fußnoten erhalten. Eine CSS-Regel im Stylesheet stellt die Fußnoten kursiv dar, sodass ein betroffener Absatz wie in der folgenden Abbildung aussieht:

How admirable is the Law of Compensation! *And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!* By a judicious use of this Law of Nature, the Polygons and Circles are almost always able to stifle sedition in its very cradle, taking advantage of the irrepressible and boundless hopefulness of the human mind....

Jetzt müssen wir die Fußnoten entnehmen und an den unteren Rand des Dokuments verschieben. Genauer gesagt, fügen wir sie zwischen `<div class="chapter">` und `<div id="footer">` ein.

Denken Sie daran, dass selbst bei impliziter Iteration die Reihenfolge, in der die Elemente verarbeitet werden, genau festgelegt ist. Sie beginnt an der Spitze des DOM-Baums und arbeitet sich nach unten durch. Da es wichtig ist, dass die korrekte Reihenfolge der Fußnoten an ihrem neuen Platz auf der Seite erhalten bleibt, verwenden wir `.insertBefore('#footer')`, wie Sie im folgenden Codeausschnitt sehen können. Dadurch wird jede Fußnote unmittelbar vor `<div id="footer">` platziert, sodass Fußnote 1 zwischen `<div class="chapter">` und `<div id="footer">` landet, Fußnote 2 zwischen Fußnote 1 und `<div id="footer">` usw. Hätten wir dagegen `.insertAfter('div.chapter')` verwendet, wären die Fußnoten in umgekehrter Reihenfolge erschienen.

Bis jetzt sieht unser Code wie folgt aus:

```
$(document).ready(function() {
    $('span.footnote').insertBefore('#footer');
});
```

#### **Listing 5–9**

Die Fußnoten sind von einem `<span>`-Tag umgeben, sodass sie standardmäßig in einer Zeile angezeigt werden, eine unmittelbar hinter der anderen. Deswegen haben wir im CSS-Code den `span.footnote`-Elementen aber schon vorausschauend den `display`-Wert `block` gegeben, wenn sie sich außerhalb von `<div class="chapter">` befinden. Daher nehmen unsere Fußnoten jetzt so langsam Form an:

to mutual warfare, and perish by one another's angles. No less than  
one hundred and twenty rebellions are recorded in our annals, besides  
minor outbreaks numbered at two hundred and thirty-five; and they  
have all ended thus.

[back to top](#)

*"What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular.*

*The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level.*

*And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!*

Die Fußnoten befinden sich jetzt zwar in der richtigen Position, aber es gibt noch mehr zu tun. Eine vollständige Lösung für Fußnoten muss auch folgende Aspekte berücksichtigen:

1. Die Fußnoten müssen nummeriert sein.
2. Die Stelle im Text, an der die Fußnote entnommen wurde, muss mit der Nummer der Fußnote markiert sein.
3. Es muss einen Link von der Textstelle zur zugehörigen Fußnote und von der Fußnote zurück zu der Textstelle geben.

### 5.2.5 Elemente verschachteln

Um die Fußnoten zu nummerieren, könnten wir die Nummern explizit in das Markup schreiben. Allerdings können wir die Nummerierung auch dem Standard-element für geordnete Listen überlassen. Dazu müssen wir ein umschließendes `<ol>`-Element erstellen, das alle Fußnoten umgibt, und ein `<li>`-Element für jede einzelne Fußnote. Hier kommen uns die *Wrappermethoden* zu Hilfe.

Wenn wir Elemente in anderen Elementen schachtern, müssen wir uns genau darüber im Klaren sein, ob es einen eigenen Container für jedes Element oder einen einzigen Container für alle Elemente geben soll. Für die Fußnotennummerierung brauchen wir beide Arten von Wrappern:

```
$(document).ready(function() {  
    $('span.footnote')  
        .insertBefore('#footer')  
        .wrapAll('<ol id="notes"></ol>')  
        .wrap('<li></li>');  
});
```

#### ***Listing 5–10***

Nachdem wir die Fußnoten vor dem Footerbereich eingefügt haben, verpacken wir den ganzen Satz mithilfe von `.wrapAll()` in einem einzigen `<ol>`-Element und anschließend jede einzelne Fußnote mit `.wrap()` in ihrem eigenen `<li>`-Element. Wie Sie im folgenden Screenshot sehen, kommt dadurch eine korrekte Nummerierung der Fußnoten zustande:

Minor Outbacks numbered at the hundred and thirty five, and they have all ended thus.

[back to top](#)

---

1. "What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular.
2. The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level.
3. And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland!

Jetzt sind wir bereit, die Stelle, von der wir die Fußnote entnehmen, zu markieren und zu nummerieren. Damit dies auf einfache Weise geschehen kann, müssen wir unseren vorhandenen Code jedoch umschreiben, ohne dabei die implizite Iteration in Anspruch zu nehmen.

Die Methode `.each()`, die zur *expliziten Iteration* dient, ähnelt sehr stark einer `for`-Schleife. Wir können sie einsetzen, wenn der Code, den wir auf alle ausgewählten Elemente anwenden möchten, für die Syntax der impliziten Iteration zu kompliziert ist. Diese Methode wird als Callback-Funktion übergeben, die für jedes passende Element einmal aufgerufen wird:

```
$(document).ready(function() {
    var $notes = $('

</ol>').insertBefore('#footer');
    $('span.footnote').each(function(index) {
        $(this).appendTo($notes).wrap('<li></li>');
    });
});
```

#### ***Listing 5-11***

Warum wir diese Änderung vornehmen, wird in Kürze klar werden. Zunächst aber müssen wir verstehen, welche Informationen unserem `.each()`-Callback zur Verfügung gestellt werden.

Wie bei den anderen Callbacks, die wir kennengelernt haben – z.B. den Wert-Callbacks, mit denen wir weiter vorn in diesem Kapitel gearbeitet haben –, zeigt das Kontextschlüsselwort `this` auf das fragliche DOM-Element. In Listing 5-11 haben wir den Kontext genutzt, um ein jQuery-Objekt zu erstellen, das auf das `<span>`-Element einer einzelnen Fußnote zeigt, dieses Element an das `<ol>`-Element der Fußnoten anzuhängen und die Fußnote schließlich in ein `<li>`-Element einzupacken.

Um die Textstellen zu markieren, an denen die Fußnoten entnommen wurden, können wir den Parameter des `.each()`-Callbacks nutzen. Er stellt einen Iterationszähler bereit, der bei null beginnt und bei jedem Aufruf des Callbacks erhöht wird. Daher ist der Wert dieses Zählers immer um 1 kleiner als die Fußnotennummer. Unter Ausnutzung dieses Umstandes können wir wie folgt die passenden Markierungen im Text hervorrufen:

```
$(document).ready(function() {
    var $notes = $('

</ol>').insertBefore('#footer');
    $('span.footnote').each(function(index) {
        $('' + (index + 1) + '').insertBefore(this);
        $(this).appendTo($notes).wrap('<li></li>');
    });
});
```

#### ***Listing 5-12***

Bevor die einzelnen Fußnoten jetzt aus dem Text herausgenommen und am unteren Rand der Seite platziert werden, erstellen wir ein neues <sup>-Element mit der Fußnotennummer und fügen es in den Text ein. Dabei ist die Reihenfolge der Aktionen wichtig: Wir müssen sicherstellen, dass die Markierung vor dem Verschieben der Fußnote eingefügt wird, da wir sonst keinen Hinweis mehr auf die ursprüngliche Position haben. Beachten Sie auch, dass der Ausdruck (index + 1) in Klammern stehen muss, damit er als Addition interpretiert wird, da + in JavaScript auch zur Stringverkettung verwendet wird.

Wenn wir uns die Seite erneut ansehen, können wir jetzt die Markierungen an den Stellen erkennen, wo zuvor die Fußnoten standen:

subject of rejoicing in our country for many furlongs round. After a strict examination conducted by the Sanitary and Social Board, the infant, if certified as Regular, is with solemn ceremonial admitted into the class of Equilaterals. He is then immediately taken from his proud yet sorrowing parents and adopted by some childless Equilateral.<sup>2</sup>

[back to top](#)

How admirable is the Law of Compensation!<sup>3</sup> By a judicious use of this Law of Nature, the Polygons and Circles are almost always able to stifle sedition in its very cradle, taking advantage of the irrepressible and boundless hopefulness of the human mind....

### 5.2.6 Umgekehrte Einfügemethoden

In Listing 5–12 haben wir einen Inhalt vor einem Element eingefügt und dann dasselbe Element an einer anderen Stelle im Dokument eingehängt. Wenn wir in jQuery mit Elementen arbeiten, können wir gewöhnlich mehrere Aktionen verketten, um sie möglichst knapp und effizient auszuführen. Das ist hier jedoch nicht möglich, da `this` das Ziel von `.insertBefore()` und das »Objekt« von `.appendTo()` ist. Mithilfe der *umgekehrten Einfügemethoden* können wir diese Einschränkung jedoch überwinden.

Zu jeder der Einfügemethoden wie `.insertBefore()` und `.appendTo()` gibt es eine umgekehrte Methode, die genau dieselbe Aufgabe ausführt, aber die Rolle von »Objekt« und »Ziel« vertauscht. Betrachten Sie dazu das folgende Beispiel:

```
$( '<p>Hello</p>' ).appendTo( '#container' );
```

Dies ist dasselbe wie:

```
$( '#container' ).append( '<p>Hello</p>' );
```

Durch die Verwendung von `.before()`, der umgekehrten Form von `.insertBefore()`, können wir unseren Code umschreiben und dadurch die Verkettung nutzen:

```
$(document).ready(function() {
    var $notes = $('<ol id="notes"></ol>')
        .insertBefore('#footer');
    $('span.footnote').each(function(index) {
        $(this)
            .before('<sup>' + (index + 1) + '</sup>')
            .appendTo($notes)
            .wrap('<li></li>');
    });
});
```

**Listing 5–13**

### Callbacks für Einfügemethoden

Die umgekehrten Einfügemethoden können ebenso wie `.attr()` und `.css()` Funktionen als Argumente annehmen. Diese Funktionen werden einmal für jedes Zielement aufgerufen und sollten den einzufügenden HTML-String zurückgeben. Diese Technik könnten wir zwar auch hier verwenden, doch treten bei jeder Fußnote mehrere solcher Situationen auf, weshalb ein einziger `.each()`-Aufruf letzten Endes die sauberere Lösung darstellt.

Jetzt können wir uns an den letzten Punkt unserer Checkliste machen, nämlich einen Link von der Textstelle zur zugehörigen Fußnote und von dort zurück zu der Textstelle vorsehen. Dazu brauchen wir vier Markup-Elemente für jede Fußnote: zwei Links, nämlich einen im Text und einen in der Fußnote, und zwei `id`-Attribute an diesen Stellen. Da das Argument der Methode `.before()` allmählich zu kompliziert zu werden droht, ist dies eine gute Gelegenheit, um eine neue Schreibweise zum Erstellen von Strings einzuführen.

In Listing 5–13 haben wir unsere Fußnotenmarkierung mit dem Operator `+` vorbereitet, der zur Verkettung von Strings dient. Das ist durchaus eine sinnvolle Technik, aber wenn wir eine große Menge von Strings zusammenfügen müssen, kann das sehr unübersichtlich aussehen. Stattdessen können wir zur Konstruktion des größeren Strings die Array-Methode `.join()` verwenden. Die beiden folgenden Anweisungen haben beide dieselbe Wirkung:

```
var str = 'a' + 'b' + 'c';
var str = ['a', 'b', 'c'].join('');
```

In diesem Beispiel müssen wir für die Methode `.join()` zwar mehr Zeichen eingeben, aber dafür sorgt sie bei verwirrenden Mischungen von Addition und Stringverkettung für mehr Klarheit. Schreiben wir unseren Code also mit `.join()` zur Verkettung der Strings um:

```
$(document).ready(function() {
    var $notes = $('<ol id="notes"></ol>')
        .insertBefore('#footer');
```

```
$('span.footnote').each(function(index) {  
    $(this)  
        .before([  
            '<sup>',  
            index + 1,  
            '</sup>'  
        ].join(''))  
        .appendTo($notes)  
        .wrap('<li></li>');  
});  
});
```

**Listing 5–14**

Da jeder Array-Eintrag einzeln berechnet wird, benötigen wir jetzt keine Klammern mehr um `index + 1`.

Mithilfe dieser Technik können wir die Fußnotenmarkierungen mit einem Link zum unteren Seitenrand sowie mit einem `id`-Wert ausstatten. Da wir gerade dabei sind, fügen wir auch dem `<li>`-Element eine `id` hinzu, damit der Link ein Ziel hat, auf das er zeigen kann:

```
$(document).ready(function() {  
    var $notes = $('<ol id="notes"></ol>')  
        .insertBefore('#footer');  
    $('span.footnote').each(function(index) {  
        $(this)  
            .before([  
                '<a href="#footnote-' +  
                index + 1,  
                '" id="context-' +  
                index + 1,  
                '" class="context">',  
                '<sup>',  
                index + 1,  
                '</sup></a>'  
            ].join(''))  
            .appendTo($notes)  
            .wrap('<li id="footnote-' + (index + 1) + '"></li>');  
    });  
});
```

**Listing 5–15**

Mit diesem zusätzlichen Markup sind die einzelnen Fußnotenmarkierungen nun mit den zugehörigen Fußnoten am unteren Rand des Dokuments verlinkt. Jetzt müssen wir nur noch den Link zurück von der Fußnote zu deren Kontext erstellen. Dazu verwenden wir wie folgt `.append()`, die umgekehrte Form der Methode `.appendTo()`:

```
$(document).ready(function() {
    var $notes = $('<ol id="notes"></ol>')
        .insertBefore('#footer');
    $('span.footnote').each(function(index) {
        $(this)
            .before([
                '<a href="#footnote-' + index + 1,
                '" id="context-' + index + 1,
                '" class="context">',
                '<sup>',
                index + 1,
                '</sup></a>'
            ].join(''))
            .appendTo($notes)
            .append([
                '&nbsp;(<a href="#context-' + index + 1,
                '">context</a>)'
            ].join(''))
            .wrap('<li id="footnote-' + (index + 1) + '"></li>');
    });
});
```

**Listing 5-16**

Hierbei zeigt href zurück auf die id der zugehörigen Markierung. In der folgenden Abbildung sehen Sie wieder die Fußnoten, die jetzt mit den neuen Links versehen sind:

...minor Outcours numbered at two hundred and thirty five, and they  
have all ended thus.

[back to top](#)

1. "What need of a certificate?" a Spaceland critic may ask: "Is not the procreation of a Square Son a certificate from Nature herself, proving the Equal-sidedness of the Father?" I reply that no Lady of any position will marry an uncertified Triangle. Square offspring has sometimes resulted from a slightly Irregular Triangle; but in almost every such case the Irregularity of the first generation is visited on the third; which either fails to attain the Pentagonal rank, or relapses to the Triangular. ([context](#))
2. The Equilateral is bound by oath never to permit the child henceforth to enter his former home or so much as to look upon his relations again, for fear lest the freshly developed organism may, by force of unconscious imitation, fall back again into his hereditary level. ([context](#))
3. And how perfect a proof of the natural fitness and, I may almost say, the divine origin of the aristocratic constitution of the States of Flatland! ([context](#))

## 5.3 Elemente kopieren

Bis jetzt haben wir in diesem Kapitel neu erstellte Elemente eingefügt, Elemente von einer Stelle im Dokument zu einer anderen verschoben und vorhandene Elemente in neue eingepackt. Manchmal müssen wir Elemente jedoch auch kopieren. Beispielsweise können wir ein Navigationsmenü, das im Kopf der Seite erscheint, kopieren und zusätzlich im Fußbereich platzieren. Wann immer wir Elemente kopieren wollen, um die Seite grafisch aufzuwerten, können wir dazu jQuery nutzen. Warum sollten wir auch etwas zweimal schreiben und damit die Wahrscheinlichkeit für Fehler verdoppeln, wenn wir es auch einmal schreiben und jQuery die weitere Arbeit überlassen können?

Die jQuery-Methode `.clone()` ist genau das, was wir zum Kopieren von Elementen brauchen. Sie nimmt eine beliebige Menge ausgewählter Elemente entgegen und erstellt davon eine Kopie zur späteren Verwendung. Wie bei der Elementerstellung durch die Funktion `$()`, die wir uns weiter vorn in diesem Kapitel angesehen haben, erscheinen die kopierten Elemente erst dann im Dokument, wenn wir eine der Einfügemethoden angewandt haben.

Mit der folgenden Zeile erstellen wir beispielsweise eine Kopie des ersten Absatzes innerhalb von `<div class="chapter">`:

```
($('div.chapter p:eq(0)').clone();
```

Das allein reicht aber nicht aus, um den Inhalt der Seite zu ändern. Mit einer Einfügemethode können wir jedoch dafür sorgen, dass der geklonte Absatz vor `<div class="chapter">` erscheint:

```
($('div.chapter p:eq(0)').clone().insertBefore('div.chapter'));
```

Dadurch erscheint der erste Absatz zweimal. Um ein Bild aus einem anderen, vertrauten Bereich zu geben: `.clone()` verhält sich zu den Einfügemethoden wie das Kopieren (Copy) zum Einfügen (Paste).

### Klonen mit Ereignissen

Standardmäßig kopiert die Methode `.clone()` keine Ereignisse, die an das ausgewählte Element oder einen seiner Nachfahren gebunden sind. Sie nimmt jedoch einen booleschen Parameter an und kopiert die Ereignisse mit, wenn dieser Parameter auf `true` gesetzt ist (`.clone(true)`). Dank dieser komfortablen Klonmethode für Ereignisse müssen wir uns nicht mit der manuellen Neubindung von Ereignissen beschäftigen, wie sie in Kapitel 3 besprochen wurde.

### 5.3.1 Klonen für interne Zitate

Auf vielen Websites werden ebenso wie in Printmedien interne Zitate, sogenannte Pull-Quotes, verwendet, um kleine Teile des Textes hervorzuheben und den Blick des Lesers darauf zu lenken. Dabei handelt es sich einfach um einen Auszug aus dem Hauptdokument, der grafisch aufgewertet neben dem Text steht. Eine solche Ausschmückung können wir mit der Methode `.clone()` auf einfache Weise erreichen. Als Erstes schauen wir uns den dritten Absatz unseres Beispieltextes noch einmal genauer an:

```
<p>
  <span class="pull-quote">
    It is a Law of Nature
    <span class="drop">
      with us
    </span>
    that a male child shall have
    <strong>
      one more side
    </strong>
    than his father
  </span>,
  so that each generation shall rise (as a rule) one step in
  the scale of development and nobility. Thus the son of a
  Square is a Pentagon; the son of a Pentagon, a Hexagon; and
  so on.
</p>
```

Wie Sie sehen, beginnt der Absatz mit `<span class="pull-quote">`. Dies ist die Klasse, die wir als Ziel für das Klonen verwenden. Nachdem wir den kopierten Text innerhalb dieses `<span>`-Bereichs an einer anderen Stelle eingefügt haben, müssen wir seine Formateigenschaften ändern, um ihn grafisch vom Rest des Textes abzusetzen.

Um diese Art der Formatierung zu erreichen, fügen wir dem kopierten `<span>`-Bereich die Klasse `pulled` hinzu. In unserem Stylesheet gibt es für diese Klasse die folgende Formatregel:

```
.pulled {
  background: #e5e5e5;
  position: absolute;
  width: 145px;
  top: -20px;
  right: -180px;
  padding: 12px 5px 12px 10px;
  font: italic 1.4em "Times New Roman", Times, serif;
}
```

Ein Element dieser Klasse wird mit einem hellgrauen Hintergrund, einem Innenrand und einer anderen Schriftart versehen. Vor allem aber wird es absolut positioniert, 20 Pixel oberhalb und 20 Pixel rechts vom nächsten (absolut oder relativ) positionierten Vorfahren im DOM. Verfügt keiner der Vorfahren über eine andere Positionierung als static, wird das interne Zitat relativ zum `<body>`-Element des Dokuments platziert. Daher müssen wir im jQuery-Code sicherstellen, dass für das Elternelement des geklonten internen Zitats `position: relative` festgelegt ist.

### Berechnungen für die CSS-Positionierung

Während die Positionierung nach oben ziemlich einleuchtend ist, mag es auf den ersten Blick nicht unbedingt klar sein, wie der Kasten mit dem hervorgehobenen internen Zitat 20 Pixel rechts von dem positionierten Elternelement platziert wird. Als Erstes bestimmen wir die Gesamtbreite des Zitatkastens, die sich aus dem Wert der Eigenschaft `width` zuzüglich des linken und rechten Innenrands ergibt, also 145 Pixel + 5 Pixel + 10 Pixel = 160 Pixel. Anschließend legen wir die Eigenschaft `right` des Zitats fest. Ein Wert von 0 würde den rechten Rand des Zitats bündig mit dem des Elternelements ausrichten. Damit der linke Rand 20 Pixel vom rechten Rand des Elternelements entfernt steht, müssen wir es in negativer Richtung um 20 Pixel mehr verschieben, als seine Gesamtbreite beträgt. So ergibt sich der Wert `-180px`.

Als Nächstes überlegen wir, welcher jQuery-Code erforderlich ist, um dieses Format anzuwenden. Wir beginnen mit einem Selektorausdruck, um alle `<span class="pull-quote">`-Elemente zu finden, und wenden dann wie bereits erwähnt das Format `position: relative` auf die jeweiligen Elternelemente an:

```
$(document).ready(function() {
    $('span.pull-quote').each(function(index) {
        var $parentParagraph = $(this).parent('p');
        $parentParagraph.css('position', 'relative');
    });
});
```

#### **Listing 5–17**

Da wir später noch einmal auf den Elternabsatz verweisen müssen, sorgt die Definition der Variablen `$parentParagraph` für eine Verbesserung der Performance und der Lesbarkeit.

Als Nächstes müssen wir das hervorgehobene Zitat erstellen, wobei wir den vorbereiteten CSS-Code nutzen. Wir klonen die einzelnen `<span>`-Elemente, fügen den Kopien die Klasse `pulled` hinzu und platzieren sie an den Anfang des Elternabsatzes:

```
$(document).ready(function() {
    $('span.pull-quote').each(function(index) {
        var $parentParagraph = $(this).parent('p');
        $parentParagraph.css('position', 'relative');

        var $clonedCopy = $(this).clone();
        $clonedCopy
            .addClass('pulled')
            .prependTo($parentParagraph);
    });
});
```

**Listing 5–18**

Wir führen auch hier mit `$clonedCopy` eine Variable ein, die sich später als nützlich erweisen wird.

Da wir das Zitat absolut positionieren, ist seine Platzierung innerhalb des Absatzes irrelevant. Solange es innerhalb des Absatzes bleibt, wird es gemäß den CSS-Regeln relativ zu dessen oberen und rechten Rand positioniert.

Jetzt erscheint das interne Zitat wie beabsichtigt neben dem Originaltext:

of all.

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

But this rule applies not always to the Tradesman, and still less often to the Soldiers, and to the Workmen; who indeed can hardly be said

*It is a Law of Nature with us that a male child shall have **one more side** than his father*

Das ist ein guter Anfang. Mit unseren nächsten Verbesserungen werden wir den Inhalt des Zitats etwas vereinfachen.

## 5.4 Get- und Set-Methoden für Inhalte

Es wäre schön, wenn wir den Inhalt des internen Zitats ein wenig ändern könnten, indem wir einige Worte entfernen und durch Auslassungspunkte ersetzen, sodass der Inhalt kürzer wird. Dazu haben wir einige der Worte im Beispieltext in ein `<span class="drop">`-Tag eingeschlossen.

Die einfachste Möglichkeit, um diese Ersetzung durchzuführen, besteht darin, unmittelbar das HTML-Markup anzugeben, das anstelle des alten eingefügt werden soll. Für unsere Zwecke ist die Methode `.html()` ideal geeignet, wie der folgende Codeausschnitt zeigt:

```
$(document).ready(function() {
    $('span.pull-quote').each(function(index) {
        var $parentParagraph = $(this).parent('p');
        $parentParagraph.css('position', 'relative');

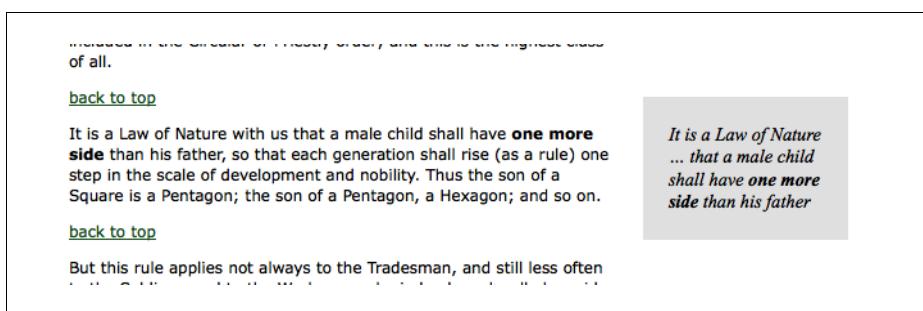
        var $clonedCopy = $(this).clone();
        $clonedCopy
            .addClass('pulled')
            .find('span.drop')
            .html('&hellip;')
        .end()
        .prependTo($parentParagraph);
    });
});
```

**Listing 5–19**

Die neuen Zeilen in Listing 5–19 stützen sich auf die Techniken zum Durchlaufen des DOM, die wir in Kapitel 2 gelernt haben. Mit `.find()` suchen wir innerhalb des Zitats nach `<span class="drop">`-Elementen, bearbeiten sie und geben dann das Zitat durch den Aufruf von `.end()` zurück. Zwischen diesen Methoden rufen wir `.html()` auf, um den Inhalt in Auslassungspunkte zu ändern (mit dem entsprechenden HTML-Zeichencode).

Wird die Methode `.html()` ohne Argumente aufgerufen, gibt sie eine Stringdarstellung des HTML-Markups im ausgewählten Element zurück. Geben wir dagegen ein Argument an, wird der Inhalt des Elements dadurch ersetzt. Achten Sie darauf, nur gültiges HTML als Argument anzugeben, wenn Sie diese Technik verwenden, und maskieren Sie alle Sonderzeichen.

Die angegebenen Worte sind jetzt durch Auslassungspunkte ersetzt worden, wie der folgende Screenshot zeigt:



Hervorgehobene interne Zitate behalten gewöhnlich nicht ihre ursprüngliche Textauszeichnung wie die Fettschrift von **one more side** in diesem Beispiel. Was wir hier wirklich anzeigen wollen, ist der Text des `<span class="pull-quote">`-Elements ohne jegliche `<strong>`-, `<em>`-, `<a href>`- oder andere Inline-Tags. Um den gesamten HTML-Text des Zitats durch reinen Text zu ersetzen, können wir `.text()` verwenden, die Begleitmethode von `.html()`.

Wie `.html()` kann `.text()` den Inhalt des ausgewählten Elements zurückgeben oder durch einen neuen String ersetzen. Anders als `.html()` jedoch ruft `.text()` immer reinen Text ab und fügt immer reinen Text ein. Wenn `.text()` Inhalte abruft, werden alle eingeschlossenen Tags ignoriert und HTML-Zeichencodes in Textzeichen umgewandelt. Beim Schreiben von Text werden Sonderzeichen wie < durch die entsprechenden HTML-Zeichencodes ersetzt:

```
$(document).ready(function() {
    $('span.pull-quote').each(function(index) {
        var $parentParagraph = $(this).parent('p');
        $parentParagraph.css('position', 'relative');

        var $clonedCopy = $(this).clone();
        $clonedCopy
            .addClass('pulled')
            .find('span.drop')
            .html('&hellip;')
            .end()
            .text($clonedCopy.text())
            .prependTo($parentParagraph);
    });
});
```

**Listing 5–20**

Wenn dieser raffinierte Code den Inhalt des Zitats mit `$clonedCopy.text()` abruft, bekommen wir einen String mit dem reinen Text ohne Tags, und wenn wir diesen Text gleich anschließend wieder in `.text()` eingeben, wird das Markup entfernt, sodass die Fettschrift aus unserem Beispiel verschwindet:

..... of all.

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

But this rule applies not always to the Tradesman, and still less often

*It is a Law of Nature ... that a male child shall have one more side than his father*

#### 5.4.1 Weitere Formatanpassungen

Um unser Beispiel abzuschließen, wollen wir unsere hervorgehobenen Zitate optisch noch ein wenig aufwerten, indem wir abgerundete Ecken und Schlagschatten hinzufügen. In modernen Browsern können solche Effekte mit erweiterten CSS-Möglichkeiten recht einfach dargestellt werden. Um eine maximale Browserkompatibilität zu erreichen, verlassen wir uns in unserem Beispiel jedoch

auf die altmodische Hintergrundbild-Technik. Aufgrund der variablen Höhe der Zitatkästen müssen wir dazu zwei Hintergrundbilder einsetzen.

Bei zwei Hintergrundbildern benötigen wir mindestens zwei Elemente, auf die wir sie anwenden können (damit es auch in allen Browsern funktioniert). Daher müssen wir ein weiteres `<div>` um den Inhalt des Zitats legen:

```
$document.ready(function() {
  $('span.pull-quote').each(function(index) {
    var $parentParagraph = $(this).parent('p');
    $parentParagraph.css('position', 'relative');

    var $clonedCopy = $(this).clone();
    $clonedCopy
      .addClass('pulled')
      .find('span.drop')
        .html('&hellip;')
      .end()
      .text($clonedCopy.text())
      .prependTo($parentParagraph)
      .addClass('rounded-top')
      .wrapInner('<div class="rounded-bottom"></div>');
  });
});
```

**Listing 5–21**

Hier verwenden wir `.wrapInner()`, eine weitere Variante der Wrappermethoden, die wir uns schon zuvor angesehen haben. Diese Methode fügt das neue Element innerhalb des Elements hinzu, auf das sie angewandt wird, aber so, dass es den gesamten Inhalt dieses Elements umschließt. Das neue `<div>` versorgt uns mit einer der benötigten Klassen, und die andere fügen wir mit einem einfachen Aufruf von `.addClass()` hinzu. Wir könnten das mit dem bereits bestehenden `.addClass()`-Aufruf kombinieren, hier aber halten wir diese beiden Vorgänge getrennt, um den Code deutlicher zu gruppieren.

Unsere CSS-Regeln wenden die passenden Hintergrundbilder auf die Elemente mit den Klassen `rounded-top` und `rounded-bottom` an. Wie Sie im folgenden Screenshot sehen, ist das Ergebnis recht ansprechend:

Included in the Circulus of Presay order, and this is the highest class of all.

[back to top](#)

It is a Law of Nature with us that a male child shall have **one more side** than his father, so that each generation shall rise (as a rule) one step in the scale of development and nobility. Thus the son of a Square is a Pentagon; the son of a Pentagon, a Hexagon; and so on.

[back to top](#)

But this rule applies not always to the Tradesman, and still less often

*It is a Law of Nature ... that a male child shall have one more side than his father*

## 5.5 Methoden zur DOM-Bearbeitung – kurz und bündig

jQuery bietet eine Vielzahl von Methoden zur DOM-Bearbeitung, die sich nach ihren Aufgaben und nach dem Zielort unterscheiden. Wir haben hier nicht alle behandelt, aber die meisten verhalten sich ähnlich wie die bereits gezeigten. Weitere Methoden besprechen wir in Kapitel 12. Die folgende Übersicht mag als Erinnerungsstütze dafür dienen, welche Methode sich für welche Aufgabe eignet:

1. Um neue Elemente aus dem HTML-Markup zu erstellen, verwenden Sie die Funktion `$( )`.
2. Um neue Elemente in jedes ausgewählte Element einzufügen, verwenden Sie:
  - `.append()`
  - `.appendTo()`
  - `.prepend()`
  - `.prependTo()`
3. Um neue Elemente vor oder hinter allen ausgewählten Elementen einzufügen, verwenden Sie:
  - `.after()`
  - `.insertAfter()`
  - `.before()`
  - `.insertBefore()`
4. Um neue Elemente um jedes ausgewählte Element herum einzufügen, verwenden Sie:
  - `.wrap()`
  - `.wrapAll()`
  - `.wrapInner()`
5. Um jedes ausgewählte Element durch ein neues Element oder durch Text zu ersetzen, verwenden Sie:
  - `.html()`
  - `.text()`
  - `.replaceAll()`
  - `.replaceWith()`
6. Um Elemente innerhalb jedes ausgewählten Elements zu entfernen, verwenden Sie:
  - `.empty()`
7. Um jedes ausgewählte Element und seine Nachfahren aus dem Dokument zu entfernen, ohne sie zu löschen, verwenden Sie:
  - `.remove()`
  - `.detach()`

## 5.6 Zusammenfassung

In diesem Kapitel haben wir mithilfe der jQuery-Methoden zur DOM-Bearbeitung Inhalte erstellt, kopiert, neu angeordnet und ausgeschmückt. Diese Methoden haben wir auf eine einzelne Webseite angewandt, um eine Reihe normaler Absätze in einen schön gestalteten Text mit Fußnoten, hervorgehobenen internen Zitaten und Links umzuwandeln.

Als Nächstes begeben wir uns mithilfe der Ajax-Methoden von jQuery auf die Reise zum Server.

### 5.6.1 Literatur

Das Thema DOM-Bearbeitung wird ausführlicher in Kapitel 12 behandelt. Eine vollständige Liste der verfügbaren Methoden zur DOM-Bearbeitung finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## 5.7 Übungsaufgaben

Um die folgenden Übungen durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Ändern Sie den Code zur Einführung der BACK TO TOP-Links, sodass die Links erst nach dem vierten Absatz in Erscheinung treten.
2. Fügen Sie bei einem Klick auf einen der BACK TO TOP-Links einen neuen Absatz hinter dem Link ein, der die Meldung »Sie waren hier« enthält. Sorgen Sie dafür, dass der Link nach wie vor funktioniert.
3. Stellen Sie den Autorennamen in Fettschrift dar, wenn darauf geklickt wird (fügen Sie dazu ein Tag hinzu, anstatt die Klassen oder CSS-Attribute zu bearbeiten).
4. Schwierig: Entfernen Sie bei einem weiteren Klick auf den Autorennamen das hinzugefügte `<b>`-Element (sodass zwischen Fettschrift und normalem Text umgeschaltet wird).
5. Schwierig: Fügen Sie allen Absätzen des Kapitels die Klasse `inhabitants` hinzu, ohne `.addClass()` aufzurufen. Stellen Sie dabei sicher, dass alle vorhandenen Klassen beibehalten werden.



## 6 Daten mit Ajax senden

Der Begriff *Ajax* wurde 2005 von Jesse James Garrett als Abkürzung für *Asynchronous JavaScript and XML* geprägt. Seitdem bezeichnet er jedoch viele verschiedene Dinge, da er eine Gruppe zusammenhängender Fähigkeiten und Techniken beschreibt. Eine Ajax-Lösung der einfachsten Art schließt die folgenden Elemente ein:

- JavaScript zum Auffangen von Benutzeraktionen oder anderen Browserereignissen
- Das Objekt XMLHttpRequest, das es ermöglicht, Anfragen an den Server zu stellen, ohne andere Browseraufgaben zu unterbrechen.
- Eine Datei auf dem Server, die Text in einem Datenformat wie XML, HTML oder JSON darstellt.
- Noch mehr JavaScript, um die vom Server kommenden Daten zu interpretieren und auf der Seite darzustellen.

Ajax wurde als Retter des Web angepriesen, der statische Webseiten in interaktive Webanwendungen verwandelt. Viele Frameworks sind entstanden, um den Entwicklern zu helfen, da die Implementierung des Objekts XMLHttpRequest in den verschiedenen Browsern voneinander abweicht. Zu diesen Frameworks gehört auch jQuery.

Schauen wir uns an, ob Ajax wirklich Wunder vollbringen kann.

### 6.1 Daten bei Bedarf laden

Wenn wir den ganzen Hype und das Drumherum beiseiteschieben, ist Ajax einfach nur eine Möglichkeit, um Daten vom Server zum Webbroweser oder Client zu übertragen, ohne dass dazu eine sichtbare Aktualisierung der Seite notwendig wird. Die Daten können verschiedene Formen annehmen, und wir haben viele Möglichkeiten, sie weiterzuverarbeiten, wenn sie ankommen. Das lässt sich gut erkennen, wenn wir dieselbe grundlegende Aufgabe auf unterschiedliche Weise durchführen.

Wir werden eine Seite aufbauen, die die Einträge eines Wörterbuchs nach dem Anfangsbuchstaben geordnet anzeigt. Das HTML-Markup, das den Inhaltsbereich der Seite definiert, sieht wie folgt aus:

```
<div id="dictionary">
</div>
```

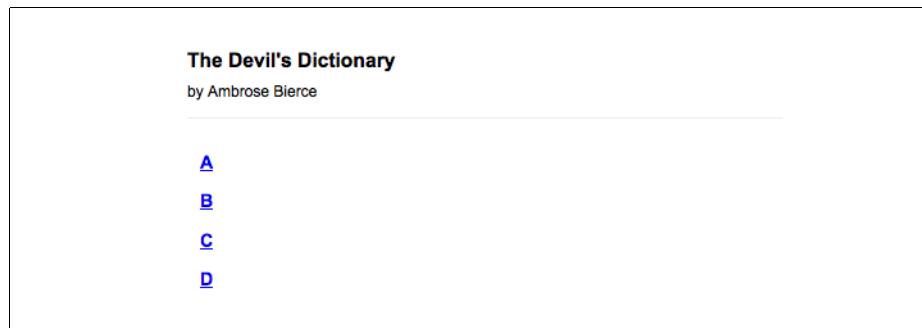
Ja, wirklich! Unsere Seite hat zu Anfang keinen Inhalt. Um dieses `<div>`-Element mit den Wörterbucheinträgen zu füllen, verwenden wir die verschiedenen Ajax-Methoden von jQuery.

Da wir eine Möglichkeit benötigen, um den Ladevorgang auszulösen, müssen wir einige Links hinzufügen, in die sich unsere Ereignishandler einklinken können:

```
<div class="letters">
  <div class="letter" id="letter-a">
    <h3><a href="#">A</a></h3>
  </div>
  <div class="letter" id="letter-b">
    <h3><a href="#">B</a></h3>
  </div>
  <div class="letter" id="letter-c">
    <h3><a href="#">C</a></h3>
  </div>
  <div class="letter" id="letter-d">
    <h3><a href="#">D</a></h3>
  </div>
  <!-- and so on -->
</div>
```

Wie immer würden wir bei einer Implementierung in der Praxis die *fortschreitende Verbesserung* nutzen, damit die Seite auch ohne JavaScript funktioniert. Um unser Beispiel zu vereinfachen, funktionieren die Links hier jedoch erst dann, wenn wir ihnen mit jQuery Verhaltensweisen hinzufügen.

Durch Einfügen einiger weniger CSS-Regeln erhalten wir eine Seite, die wie im folgenden Screenshot aussieht:



Als Nächstes kümmern wir uns darum, wie wir den Inhalt auf die Seite bekommen.

### 6.1.1 HTML anhängen

Ajax-Anwendungen sind häufig nicht mehr als die Anforderung eines großen Stücks HTML-Text. Die Umsetzung dieser Technik, die manchmal auch als *AHAH* bezeichnet wird (*Asynchronous HTTP and HTML*), ist in jQuery eine fast triviale Aufgabe. Als Erstes benötigen wir HTML-Text zum Einfügen. Wir platzieren ihn außerhalb unseres Hauptdokuments in der Datei `a.html`, die wie folgt beginnt:

```
<div class="entry">
    <h3 class="term">ABDICTION</h3>
    <div class="part">n.</div>
    <div class="definition">
        An act whereby a sovereign attests his sense of the high
        temperature of the throne.
        <div class="quote">
            <div class="quote-line">Poor Isabella's Dead, whose
                abdication</div>
            <div class="quote-line">Set all tongues wagging in the
                Spanish nation.</div>
            <div class="quote-line">For that performance 'twere
                unfair to scold her:</div>
            <div class="quote-line">She wisely left a throne too
                hot to hold her.</div>
            <div class="quote-line">To History she'll be no royal
                riddle &mdash;</div>
            <div class="quote-line">Merely a plain parched pea that
                jumped the griddle.</div>
            <div class="quote-author">G.J.</div>
        </div>
    </div>
</div>
<div class="entry">
    <h3 class="term">ABSOLUTE</h3>
    <div class="part">adj.</div>
    <div class="definition">
        Independent, irresponsible. An absolute monarchy is one
        in which the sovereign does as he pleases so long as he
        pleases the assassins. Not many absolute monarchies are
        left, most of them having been replaced by limited
        monarchies, where the sovereign's power for evil (and for
        good) is greatly curtailed, and by republics, which are
        governed by chance.
    </div>
</div>
```

Die Seite geht mit weiteren Einträgen in dieser HTML-Struktur weiter. Für sich allein dargestellt, sieht die Seite sehr schlicht aus:

## ABDICTION

n.

An act whereby a sovereign attests his sense of the high temperature of the throne.

Poor Isabella's Dead, whose abdication

Set all tongues wagging in the Spanish nation.

For that performance 'twere unfair to scold her:

She wisely left a throne too hot to hold her.

To History she'll be no royal riddle —

Merely a plain parched pea that jumped the griddle.

G.J.

## ABSOLUTE

adj.

Independent, irresponsible. An absolute monarchy is one in which the sovereign does as he pleases so long as he pleases the assassins. Not many absolute monarchies are left, most of them having been replaced by limited monarchies, where the sovereign's power for evil (and for good) is greatly curtailed, and by republics, which are governed by chance.

## ACKNOWLEDGE

Beachten Sie, dass a.html kein echtes HTML-Dokument ist, da es keines der Tags <html>, <head> und <body> enthält, die normalerweise erforderlich sind. Eine solche Datei nennen wir einen *Ausschnitt* oder ein *Fragment*. Ihr einziger Zweck besteht darin, in ein anderes HTML-Dokument eingefügt zu werden, was wir wie folgt bewerkstelligen:

```
$(document).ready(function() {
    $('#letter-a a').click(function() {
        $('#dictionary').load('a.html');
        return false;
    });
});
```

### **Listing 6-1**

Die Schwerarbeit erledigt die Methode .load() für uns. Wir geben nur den Zielort für das HTML-Fragment mithilfe eines normalen jQuery-Selektors an und übergeben dieser Methode die URL der zu ladenden Datei als Parameter. Wenn der Benutzer jetzt auf den ersten Link klickt, wird die Datei geladen und innerhalb von <div id="dictionary"> platziert. Der Browser stellt den neuen HTML-Text dar, sobald er eingefügt ist, wie der folgende Screenshot zeigt:

The Devil's Dictionary  
by Ambrose Bierce

**A** **ABDICTION** *n.*  
An act whereby a sovereign attests his sense of the high temperature of the throne.

**B** Poor Isabella's Dead, whose abdication  
Set all tongues wagging in the Spanish nation.  
For that performance 'twere unfair to scold her:  
She wisely left a throne too hot to hold her.  
To History she'll be no royal riddle —  
Merely a plain parched pea that limped the gridle

**C**

**D**

Der zuvor schlicht dargestellte HTML-Text ist jetzt formatiert. Das kommt durch die CSS-Regeln im Hauptdokument zustande. Sobald das neue HTML-Fragment eingefügt ist, gelten die Regeln auch für seine Elemente.

Beim Testen dieses Beispiels erscheinen die Definitionen des Wörterbuchs wahrscheinlich sofort nach dem Klick auf die Schaltfläche. Hier zeigt sich die Gefahr, lokal mit unseren Anwendungen zu arbeiten: Es ist schwer, Verzögerungen oder Unterbrechungen bei der Übertragung der Dokumente über ein Netzwerk zu berücksichtigen. Mit dem folgenden Code können wir eine Meldung anzeigen, nachdem die Definitionen geladen sind:

```
$(document).ready(function() {  
    $('#letter-a a').click(function() {  
        $('#dictionary').load('a.html');  
        alert('Loaded!');  
        return false;  
    });  
});
```

**Listing 6-2**

Aus der Struktur dieses Codes könnten wir folgern, dass die Meldung erst dann angezeigt wird, wenn der Ladevorgang abgeschlossen ist. JavaScript-Befehle werden gewöhnlich *synchron* ausgeführt, also eine Aufgabe nach der anderen in strikter Reihenfolge.

Wenn wir diesen Code auf einem Produktionswebserver ausprobieren und der Ladevorgang aufgrund von Verzögerungen im Netzwerk länger dauert, kann es jedoch sein, dass die Meldung bei seiner Beendigung längst angezeigt wurde und wieder verschwunden ist. Das liegt daran, dass Ajax-Aufrufe standardmäßig *asynchron* erfolgen – sonst müsste es ja auch »Sjax« heißen, was längst nicht so vielversprechend klingt!

Asynchrones Laden bedeutet, dass die Skriptausführung unmittelbar wieder aufgenommen wird, nachdem die HTTP-Requests zum Abruf des HTML-Fragments gesendet wurden. Irgendwann später empfängt der Browser dann die Ant-

wort vom Server und verarbeitet sie. Das ist im Allgemeinen das erwünschte Verhalten. Es wäre ungünstig, den ganzen Webbrowser einzufrieren, während er auf den Empfang der Daten wartet.

Um Aktionen verzögern zu können, bis der Ladevorgang abgeschlossen ist, bietet jQuery einen *Callback*. Callbacks haben wir bereits in Kapitel 4, »Formatierung und Animation«, verwendet, um Aktionen nach dem Abschluss eines Effekts auszuführen. Ajax-Callbacks erfüllen eine ähnliche Funktion, da sie ausgeführt werden, nachdem die Daten vom Server angekommen sind. Ein Beispiel dafür sehen Sie im folgenden Abschnitt.

### 6.1.2 Mit JavaScript-Objekten arbeiten

Vollständig formatierten HTML-Text bei Bedarf abzurufen, ist sehr praktisch, bedeutet aber, dass wir zusammen mit dem eigentlichen Inhalt eine ganze Menge an Informationen über die HTML-Struktur übertragen müssen. Es gibt jedoch Situationen, in denen wir lieber so wenig Daten wie möglich übertragen und sie nach ihrer Ankunft verarbeiten möchten. In diesem Fall müssen wir die Daten in einer Struktur abrufen, die wir mit JavaScript durchlaufen können.

Mit den jQuery-Selektoren können wir den erhaltenen HTML-Text zwar durchlaufen und bearbeiten, aber wenn wir ein JavaScript-ähnliches Datenformat verwenden, müssen wir weniger Daten übertragen und weniger Code verarbeiten.

#### JSON-Daten abrufen

Wie wir schon häufig gesehen haben, sind *JavaScript-Objekte* nichts anderes als Mengen von *Schlüssel-Wert-Paaren*, die in knapper Form mithilfe von geschweiften Klammern ({} ) definiert werden. *JavaScript-Arrays* dagegen werden im laufenden Betrieb mit eckigen Klammern ([]) definiert und haben implizite Schlüssel, bei denen es sich um fortlaufende, ganzzählige Nummern handelt. Durch die Kombination dieser beiden Prinzipien können wir sehr vielschichtige Datenstrukturen ausdrücken.

Der Begriff *JavaScript Object Notation (JSON)* wurde von Douglas Crockford geprägt, um diese einfache Syntax auszunutzen. Diese Schreibweise bietet eine knappe Alternative zu dem manchmal doch etwas sperrigen XML-Format, wie der folgende Codeausschnitt zeigt:

```
{  
    "key": "value",  
    "key 2": [  
        "array",  
        "of",  
        "items"  
    ]  
}
```

JSON basiert zwar auf den *Objekt- und Array-Literalen* von JavaScript, weist aber strengere Vorschriften für die Syntax und mehr Einschränkungen für die zugelassenen Werte auf. Beispielsweise verlangt JSON, dass alle Objektschlüssel ebenso wie alle Stringwerte in doppelte Anführungszeichen eingeschlossen sein müssen. Darüber hinaus sind Funktionen keine gültigen JSON-Werte. Wegen dieser strengen Regeln sollten Entwickler JSON-Daten besser nicht manuell bearbeiten, sondern sich für eine korrekte Formatierung lieber auf eine serverseitige Sprache verlassen.

Informationen über die Syntaxanforderungen von JSON, ihre möglichen Vorteile und die Implementierung in vielen Programmiersprachen finden Sie unter <http://json.org/>.

Unsere Daten können wir auf verschiedene Weise in diesem Format codieren. Wir platzieren einige Wörterbucheinträge in einer JSON-Datei, die wir b.json nennen:

```
[  
  {  
    "term": "BACCHUS",  
    "part": "n.",  
    "definition": "A convenient deity invented by the...",  
    "quote": [  
      "Is public worship, then, a sin,",  
      "That for devotions paid to Bacchus",  
      "The lictors dare to run us in,",  
      "And resolutely thump and whack us?"  
    ],  
    "author": "Jorace"  
  },  
  {  
    "term": "BACKBITE",  
    "part": "v.t.",  
    "definition": "To speak of a man as you find him when..."  
  },  
  {  
    "term": "BEARD",  
    "part": "n.",  
    "definition": "The hair that is commonly cut off by..."  
  },  
  ... file continues ...
```

Um diese Daten abzurufen, verwenden wir die Methode `$.getJSON()`, die die Datei nimmt und verarbeitet. Wenn die Daten vom Server eintreffen, haben sie die Form eines Textstrings im JSON-Format. Die Methode `$.getJSON()` analysiert diesen String und stellt dem aufrufenden Code das resultierende JavaScript-Objekt zur Verfügung.

## Globale jQuery-Funktionen

Alle bis jetzt verwendeten Query-Methoden wurden mit einem jQuery-Objekt verknüpft, das wir mit der Funktion `$( )` erstellt haben. Mit den Selektoren konnten wir einen Satz von DOM-Knoten auswählen, die dann auf irgendeine Weise von diesen Methoden bearbeitet wurden. Die Funktion `$.getJSON()` jedoch ist anders geartet. Es gibt kein DOM-Element, auf das sie logischerweise angewendet werden könnte, und das resultierende Objekt muss dem Skript bereitgestellt und nicht auf der Seite eingefügt werden. Daher ist `getJSON()` als eine Methode des *globalen jQuery-Objekts* definiert (eines einzelnen Objekts namens `jQuery` oder `$`, das einmalig von der Bibliothek `jQuery` definiert wird) und nicht als die einer einzelnen *Instanz des jQuery-Objekts* (des Objekts, das die Funktion `$( )` zurückgibt).

Wenn JavaScript wie andere objektorientierte Sprachen über Klassen verfügte, würden wir `$.getJSON()` als eine Klassenmethode bezeichnen. Hier nennen wir diese Art von Methode eine *globale Funktion*. Tatsächlich handelt es sich dabei um eine Funktion, die den *Namensraum* `jQuery` verwendet, damit es nicht zu Konflikten mit anderen Funktionsnamen kommt.

Um diese Funktion zu verwenden, übergeben wir ihr wie zuvor den Dateinamen:

```
// Unfinished code
$(document).ready(function() {
    $('#letter-b a').click(function() {
        $.getJSON('b.json');
        return false;
    });
});
```

**Listing 6-3**

Wenn wir auf den Link klicken, zeigt dieser Code keine sichtbare Wirkung. Der Funktionsaufruf lädt zwar die Datei, aber wir haben JavaScript nicht angewiesen, was mit den resultierenden Daten geschehen soll. Dazu brauchen wir eine *Callback-Funktion*.

Als zweites Argument nimmt `$.getJSON()` eine Funktion an, die beim Abschluss des Ladevorgangs aufgerufen wird. Da Ajax-Aufrufe wie bereits erwähnt asynchron sind, bietet dieser Callback eine Möglichkeit, um auf die Übertragung der Daten zu warten, anstatt den Code sofort auszuführen. Die Callback-Funktion nimmt ebenfalls ein Argument entgegen, das mit den resultierenden Daten ausgefüllt wird. Damit können wir folgenden Code schreiben:

```
// Unfinished code
$(document).ready(function() {
    $('#letter-b a').click(function() {
        $.getJSON('b.json', function(data) {
    });
});
```

```
        return false;
    });
});
});
```

**Listing 6–4**

Hier verwenden wir in unserem jQuery-Code der Einfachheit halber einen *anonymen Funktionsausdruck* als Callback. Genauso gut können wir als Callback jedoch auch einen Verweis auf eine *Funktionsdeklaration* angeben.

Innerhalb der Funktion verwenden wir die Variable `data`, um die JSON-Struktur nach Bedarf zu durchlaufen. Wir müssen über das Array der obersten Ebene iterieren und den HTML-Text für jedes Element konstruieren. Das könnten wir mit einer normalen `for`-Schleife erledigen, doch stattdessen wollen wir eine weitere der nützlichen globalen Funktionen von jQuery einführen, nämlich `$.each()`. Das Gegenstück dazu, die Methode `.each()`, haben wir bereits in Kapitel 5, »DOM-Bearbeitung«, kennengelernt. Diese Funktion wirkt sich nicht auf die jQuery-Sammlung von DOM-Elementen aus, sondern nimmt ein Array oder eine Zuordnung als ersten Parameter an und eine Callback-Funktion als zweiten. Bei jedem Schleifendurchlauf werden der aktuelle *Iterationsindex* und das aktuelle Element des Arrays oder der Zuordnung in Form zweier Parameter an die Callback-Funktion übergeben, wie der folgende Code zeigt:

```
$(document).ready(function() {
    $('#letter-b a').click(function() {
        $.getJSON('b.json', function(data) {
            var html = '';
            $.each(data, function(entryIndex, entry) {
                html += '<div class="entry">';
                html += '<h3 class="term">' + entry.term + '</h3>';
                html += '<div class="part">' + entry.part + '</div>';
                html += '<div class="definition">';
                html += entry.definition;
                html += '</div>';
                html += '</div>';
            });
            $('#dictionary').html(html);
        });
        return false;
    });
});
```

**Listing 6–5**

Wir verwenden `$.each()`, um alle Elemente nacheinander zu untersuchen und eine HTML-Struktur mit dem Inhalt der eingegebenen Zuordnung aufzubauen. Sobald der gesamte HTML-Text für jeden Eintrag erstellt ist, wird er mit `.html()` in `<div id="dictionary">` eingefügt, wo er alles ersetzt, was sich möglicherweise dort befunden hat.

## Sicheres HTML

Diese Vorgehensweise setzt voraus, dass die Daten für die HTML-Verarbeitung sicher sind. Beispielsweise dürfen sie keine vereinzelten <-Zeichen enthalten.

Jetzt müssen wir uns nur noch um die Einträge mit Zitaten kümmern, was eine weitere `$.each()`-Schleife erfordert:

```
$(document).ready(function() {
    $('#letter-b a').click(function() {
        $.getJSON('b.json', function(data) {
            var html = '';
            $.each(data, function(entryIndex, entry) {
                html += '<div class="entry">';
                html += '<h3 class="term">' + entry.term + '</h3>';
                html += '<div class="part">' + entry.part + '</div>';
                html += '<div class="definition">';
                html += entry.definition;
                if (entry.quote) {
                    html += '<div class="quote">';
                    $.each(entry.quote, function(lineIndex, line) {
                        html += '<div class="quote-line">' + line + '</div>';
                    });
                    if (entry.author) {
                        html += '<div class="quote-author">' + entry.author +
                               '</div>';
                    }
                    html += '</div>';
                }
                html += '</div>';
                html += '</div>';
            });
            $('#dictionary').html(html);
        });
        return false;
    });
});
```

### Listing 6–6

Nachdem wir diesen Code hinzugefügt haben, können wir auf den Link B klicken, um das Ergebnis zu überprüfen:

The Devil's Dictionary  
by Ambrose Bierce

A      **BACCHUS** *n.*  
A convenient deity invented by the ancients as an excuse for getting drunk.

B      Is public worship, then, a sin,  
That for devotions paid to Bacchus  
The lictors dare to run us in,  
And resolutely thump and whack us?

C      The lictors dare to run us in,  
And resolutely thump and whack us?

D      Jorace

Das JSON-Format ist kurz und bündig, verzeiht aber keine Fehler. Jede eckige und jede geschweifte Klammer, jedes Anführungszeichen und jedes Komma muss vorhanden und berücksichtigt sein, sonst wird die Datei nicht geladen. Manchmal bekommen wir nicht einmal eine Fehlermeldung – das Skript schlägt einfach stillschweigend fehl.

### Skripte ausführen

Es kann vorkommen, dass wir beim Laden der Seite nicht das gesamte JavaScript abrufen wollen, z.B. weil wir nicht wissen, welche Skripte notwendig sind, bevor der Benutzer irgendeine Aktion ausführt. Wir könnten zwar die `<script>`-Tags im laufenden Betrieb einfügen, sobald sie gebraucht werden, aber eine elegantere Möglichkeit zur Einführung von zusätzlichem Code besteht darin, die `.js`-Datei direkt von jQuery laden zu lassen.

Der Abruf eines Skripts ist ebenso einfach wie das Laden eines HTML-Fragments. In diesem Fall verwenden wir die Funktion `$.getScript()`, die wie ihre Verwandten eine URL für den Speicherort der Skriptdatei entgegennimmt:

```
$(document).ready(function() {
    $('#letter-c a').click(function() {
        $.getScript('c.js');
        return false;
    });
});
```

#### Listing 6–7

In unserem letzten Beispiel mussten wir die Ergebnisdaten verarbeiten, um mit der geladenen Datei irgendetwas Sinnvolles zu tun. Bei einer Skriptdatei dagegen erfolgt die Verarbeitung automatisch: Das Skript wird einfach ausgeführt.

Auf diese Weise abgerufene Skripte werden im *globalen Kontext* der aktuellen Seite ausgeführt. Das bedeutet, dass sie Zugriff auf alle global definierten Funktionen und Variablen haben, insbesondere auf jQuery selbst. Daher können wir das JSON-Beispiel nachahmen, um HTML-Text vorzubereiten und auf der Seite einzufügen, wenn das Skript ausgeführt wird. Dazu schreiben wir folgenden Code in `c.js`:

```
var entries = [
    {
        "term": "CALAMITY",
        "part": "n.",
        "definition": "A more than commonly plain and..."
    },
    {
        "term": "CANNIBAL",
        "part": "n.",
        "definition": "A gastronome of the old school who..."
    },
];
```

```
{  
    "term": "CHILDHOOD",  
    "part": "n.",  
    "definition": "The period of human life intermediate..."  
}  
// and so on  
];  
  
var html = '';  
  
$.each(entries, function() {  
    html += '<div class="entry">';  
    html += '<h3 class="term">' + this.term + '</h3>';  
    html += '<div class="part">' + this.part + '</div>';  
    html += '<div class="definition">' + this.definition + '</div>';  
    html += '</div>';  
});  
  
$('#dictionary').html(html);
```

Ein Klick auf den Link C führt jetzt zu dem erwarteten Ergebnis und zeigt die entsprechenden Wörterbucheinträge.

### 6.1.3 XML-Dokumente laden

Das X in der Abkürzung Ajax steht für XML, aber bis jetzt haben wir noch keine XML-Daten geladen. Dieser Vorgang ist jedoch recht einfach und ähnelt sehr stark der JSON-Technik. Als Erstes benötigen wir eine XML-Datei, d.xml, mit einigen Daten, die wir anzeigen möchten:

```
<?xml version="1.0" encoding="UTF-8"?>  
<entries>  
    <entry term="DEFAME" part="v.t.">  
        <definition>  
            To lie about another. To tell the truth about another.  
        </definition>  
    </entry>  
    <entry term="DEFENCELESS" part="adj.">  
        <definition>  
            Unable to attack.  
        </definition>  
    </entry>  
    <entry term="DELUSION" part="n.">  
        <definition>  
            The father of a most respectable family, comprising  
            Enthusiasm, Affection, Self-denial, Faith, Hope,  
            Charity and many other goodly sons and daughters.  
        </definition>  
        <quote author="Mumfrey Mappel">  
            <line>All hail, Delusion! Were it not for thee</line>  
            <line>The world turned topsy-turvy we should see;</line>
```

```
<line>For Vice, respectable with cleanly fancies,</line>
<line>Would fly abandoned Virtue's gross advances.</line>
</quote>
</entry>
</entries>
```

Diese Daten können natürlich unterschiedlich ausgedrückt werden, und einige davon würden die Struktur, die wir zuvor in HTML und JSON erstellt haben, besser nachahmen. Wir wollen hier jedoch einige der Merkmale von XML demonstrieren, die für eine bessere Lesbarkeit sorgen, z.B. die Verwendung von *Attributten* statt Tags für term und part.

Mit unserer Funktion beginnen wir auf vertraute Weise:

```
// Unfinished code
$(document).ready(function() {
    $('#letter-d a').click(function() {
        $.get('d.xml', function(data) {
            });
            return false;
        });
    });
});
```

**Listing 6–8**

Diesmal ist es die Funktion `$.get()`, die die Arbeit erledigt. Allgemein gesagt, ruft diese Funktion die Datei an der angegebenen URL ab und stellt den reinen Text für den Callback bereit. Ist allerdings aufgrund des vom Server angegebenen MIME-Typs bekannt, dass es sich bei der Antwort um XML handelt, wird dem Callback der XML-DOM-Baum übergeben.

Wie wir bereits gesehen haben, verfügt jQuery über beachtliche Fähigkeiten zum Durchlaufen des DOM. Wir können auf das XML-Dokument die normalen Methoden `.find()` und `.filter()` sowie alle anderen Traversierungsmethoden anwenden, die wir auch bei HTML einsetzen:

```
$(document).ready(function() {
    $('#letter-d a').click(function() {
        $.get('d.xml', function(data) {
            $('#dictionary').empty();
            $(data).find('entry').each(function() {
                var $entry = $(this);
                var html = '<div class="entry">';
                html += '<h3 class="term">' + $entry.attr('term');
                html += '</h3>';
                html += '<div class="part">' + $entry.attr('part');
                html += '</div>';
                html += '<div class="definition">';
                html += $entry.find('definition').text();
                var $quote = $entry.find('quote');
                if ($quote.length) {
```

```
html += '<div class="quote">';
$quote.find('line').each(function() {
    html += '<div class="quote-line">';
    html += $(this).text() + '</div>';
});
if ($quote.attr('author')) {
    html += '<div class="quote-author">';
    html += $quote.attr('author') + '</div>';
}
html += '</div>';
html += '</div>';
$('#dictionary').append($(html));
});
});
return false;
});
});
});
```

**Listing 6–9**

Ein Klick auf D führt zu dem erwarteten Ergebnis, wie der folgende Screenshot zeigt:

The Devil's Dictionary  
by Ambrose Bierce

A      **DANCE** *v.i.*  
To leap about to the sound of tittering music, preferably with arms about your neighbor's wife or daughter. There are many kinds of dances, but all those requiring the participation of the two sexes have two characteristics in common: they are conspicuously innocent, and warmly loved by the vicious.

B

C

D

Dieser neue Verwendungszweck für die Methoden zum Durchlaufen des DOM, die wir bereits kennen, wirft ein bezeichnendes Licht auf die Flexibilität, die jQuery bei der Nutzung von CSS-Selektoren bietet. CSS wird gewöhnlich eingesetzt, um HTML-Seiten aufzupolieren, weshalb die Selektoren in standardmäßigen .css-Dateien die Namen von HTML-Tags wie `div` und `body` benutzen, um Inhalte zu finden. jQuery kann jedoch nicht nur die Namen standardmäßiger HTML-Tags verwenden, sondern genauso gut auch die von beliebigen XML-Tags, wie hier `entry` und `definition`.

Das erweiterte Selektorenmodul von jQuery ermöglicht es auch, Teile eines XML-Dokuments in weit komplizierteren Situationen aufzuspüren. Nehmen wir

an, wir wollten nur die Einträge anzeigen, die über Zitate mit angegebenen Autoren verfügen. Dazu können wir die Einträge auf diejenigen mit geschachtelten <quote>-Elementen einschränken, indem wir entry in entry:has(quote) ändern. Anschließend können wir mit entry:has(quote[author]) eine weitere Eingrenzung auf Einträge mit Autorenattributen in den <quote>-Elementen vornehmen.

Die Zeile in Listing 6–9 mit dem ursprünglichen Selektor sieht jetzt wie folgt aus:

```
$ (data).find('entry:has(quote[author])').each(function() {
```

Dieser neue Selektorausdruck beschränkt die zurückgegebenen Einträge entsprechend:

The Devil's Dictionary  
by Ambrose Bierce

**A** DEBT *n.*  
An ingenious substitute for the chain and whip of the slave-driver.

**B** As, pent in an aquarium, the troutlet  
Swims round and round his tank to find an outlet,  
Pressing his nose against the glass that holds him,  
Nor ever sees the prison that enfolds him;  
So the poor debtor, seeing naught around him,  
Yet feels the narrow limits that impound him,  
Grieves at his debt and studies to evade it,  
And finds at last he might as well have paid it.

Barlow  
S. Vode

## 6.2 Ein Datenformat auswählen

Wir haben uns jetzt vier Formate für unsere externen Daten angesehen, die alle von jQuery mithilfe seiner Ajax-Funktionen gehandhabt werden können. Außerdem haben wir nachgewiesen, dass alle vier die gestellte Aufgabe erfüllen können, nämlich Informationen auf eine Seite zu laden, wenn der Benutzer sie anfordert, und nicht vorher. Wie entscheiden wir nun, welches Format wir in unserer Anwendung einsetzen?

*HTML-Fragmente* erfordern wenig Implementierungsarbeit. Die externen Daten können mit einer einfachen Methode geladen und auf der Seite eingefügt werden, wobei nicht einmal eine Callback-Funktion erforderlich ist. Für die simple Aufgabe, neuen HTML-Text auf einer vorhandenen Seite einzufügen, ist es auch nicht nötig, die Daten zu durchlaufen. Auf der anderen Seite sind die Daten nicht unbedingt so strukturiert, dass sie für andere Anwendungen wieder verwendbar sind. Die externe Datei ist eng mit dem vorgesehenen Container gekoppelt.

*JSON-Dateien* sind durch ihre Struktur für eine einfache Wiederverwendung geeignet. Sie sind kompakt und leicht zu lesen. Um die Informationen zu extrahieren und auf der Seite darzustellen, muss die Datenstruktur durchlaufen werden, aber das lässt sich mit JavaScript-Standardtechniken erledigen. Da moderne Browser die Dateien selbst mit einem einzigen Aufruf von `JSON.parse()` analysieren, werden die JSON-Dateien auch sehr schnell gelesen. Fehler in der JSON-Datei können zu einem Fehlschlag ohne jegliche Rückmeldung und sogar zu Nebenwirkungen auf der Seite führen, weshalb die Daten sorgfältig von einer vertrauenswürdigen Stelle aufbereitet werden müssen.

*JavaScript-Dateien* bieten die größtmögliche Flexibilität, sind aber eigentlich kein Mechanismus zur Datenspeicherung. Da die Dateien sprachspezifisch sind, können sie nicht dazu verwendet werden, die Informationen für verschiedene Systeme bereitzustellen. Die Möglichkeit, eine JavaScript-Datei zu laden, bedeutet aber, selten gebrauchte Verhalten in externe Dateien auslagern zu können, was die Größe des Codes verringert, bis diese Ergänzungen benötigten werden (falls überhaupt).

XML hat in der JavaScript-Community zwar an Bedeutung verloren, da die meisten Entwickler JSON bevorzugen, ist aber immer noch so gebräuchlich, dass die Bereitstellung der Daten in diesem Format die Wahrscheinlichkeit dafür erhöht, sie an anderer Stelle wiederverwenden zu können. Tatsächlich exportieren viele beliebte Social-Networking-Websites ihre Daten in XML-Darstellung, sodass es zu vielerlei interessanten *Mashups* dieser Daten kommt. Das XML-Format ist jedoch ziemlich sperrig, und die Analyse und Bearbeitung erfolgt oft langsamer als bei den anderen Möglichkeiten.

Angesichts dieser Merkmale ist es am einfachsten, externe Daten als HTML-Fragmente bereitzustellen, sofern die Daten nicht auch in anderen Anwendungen gebraucht werden. Sollen die Daten dagegen in anderen Anwendungen wieder verwendet werden, auf die man selbst Einfluss hat, ist JSON aufgrund seiner Performance und der Dateigröße meistens eine gute Wahl. Ist die andere Anwendung unbekannt, bietet XML die größte Sicherheit dafür, dass eine Zusammenarbeit möglich ist.

Am wichtigsten aber ist die Frage, ob die Daten bereits fertig bereitstehen. Wenn ja, dann liegen sie wahrscheinlich in einem der hier besprochenen Formate bereits vor, sodass uns die Entscheidung von jemand anderem abgenommen worden ist.

### 6.3 Daten an den Server übergeben

Bis jetzt ging es in unseren Beispielen darum, *statische* Datendateien vom Webserver abzurufen. Ajax läuft jedoch erst dann richtig zur Hochform auf, wenn der Server die Daten über die Eingabe vom Browser *dynamisch* gestalten kann. Auch bei dieser Aufgabe hilft uns jQuery. Alle bisher besprochenen Methoden können angepasst werden, um eine beidseitige Datenübertragung zu ermöglichen.

### Interaktion mit serverseitigem Code

Da die Demonstration dieser Techniken eine Interaktion mit dem Webserver erfordert, müssen wir hier zum ersten Mal serverseitigen Code angeben. Die folgenden Beispiele verwenden die weitverbreitete und kostenlos erhältliche Skriptsprache PHP. Wie Sie einen Webserver mit PHP einrichten, werden wir hier nicht behandeln. Hilfe dazu erhalten Sie auf den Websites von Apache (<http://apache.org/>) und PHP (<http://php.net/>) sowie beim Hosting-Provider für Ihre Website.

#### 6.3.1 GET-Requests durchführen

Um die Kommunikation zwischen Client und Server zu zeigen, schreiben wir ein Skript, das bei jeder Anforderung nur einen Wörterbucheintrag an den Browser sendet. Welcher Eintrag das ist, hängt von dem Parameter ab, den der Browser sendet. Das Skript bezieht seine Daten von einer internen Datenstruktur wie der folgenden:

```
<?php
$entries = array(
    'EAVESDROP' => array(
        'part' => 'v.i.',
        'definition' => 'Secretly to overhear a catalogue of the
                        crimes and vices of another or yourself.',
        'quote' => array(
            'A lady with one of her ears applied',
            'To an open keyhole heard, inside,',
            'Two female gossips in converse free &mdash;',
            'The subject engaging them was she.',
            '"I think," said one, "and my husband thinks',
            'That she\'s a prying, inquisitive minx!"',
            'As soon as no more of it she could hear',
            'The lady, indignant, removed her ear.',
            '"I will not stay," she said, with a pout,',
            '"To hear my character lied about!"',
        ),
        'author' => 'Gopete Sherany',
    ),
    'EDIBLE' => array(
        'part' => 'adj.',
        'definition' => 'Good to eat, and wholesome to digest, as
                        a worm to a toad, a toad to a snake, a snake to a pig,
                        a pig to a man, and a man to a worm.',
    ),
    // and so on
);
?>
```

In einer Produktionsversion dieses Beispiels wären die Daten wahrscheinlich in einer Datenbank gespeichert und würden bei Bedarf geladen. Da die Daten hier jedoch Teil des Skripts sind, ist der Code für ihren Abruf sehr einfach. Wir untersuchen die an den Server übertragenen Daten und rufen eine Funktion auf, die das anzuseigende HTML-Fragment zurückgibt:

```
<?php
if (isset($entries[strtoupper($_REQUEST['term'])])) {
    $term = strtoupper($_REQUEST['term']);
    $entry = $entries[$term];
    echo build_entry($term, $entry);
} else {
    echo '<div>Sorry, your term was not found.</div>';
}

function build_entry($term, $entry) {
    $html = '<div class="entry">';
    $html .= '<h3 class="term">';
    $html .= $term;
    $html .= '</h3>';

    $html .= '<div class="part">';
    $html .= $entry['part'];
    $html .= '</div>';

    $html .= '<div class="definition">';
    $html .= $entry['definition'];
    if (isset($entry['quote'])) {
        foreach ($entry['quote'] as $line) {
            $html .= '<div class="quote-line">' . $line . '</div>';
        }
        if (isset($entry['author'])) {
            $html .= '<div class="quote-author">' .
                $entry['author'] . '</div>';
        }
    }
    $html .= '</div>';
    $html .= '</div>';
    return $html;
}
?>
```

Anforderungen an dieses Skript, das wir e.php nennen, geben jetzt das HTML-Fragment zurück, das dem in den GET-Parametern gesendeten Begriff entspricht. Wenn wir beispielsweise mit e.php?term=eavesdrop auf das Skript zugreifen, erhalten wir irgendetwas in der Art des folgenden Screenshots zurück:

## EAVESDROP

v.i.

Secretly to overhear a catalogue of the crimes and vices of another or yourself.

A lady with one of her ears applied  
To an open keyhole heard, inside,  
Two female gossips in converse free —  
The subject engaging them was she.  
"I think," said one, "and my husband thinks  
That she's a prying, inquisitive minx!"  
As soon as no more of it she could hear  
The lady, indignant, removed her ear.  
"I will not stay," she said, with a pout,  
"To hear my character lied about!"  
Gopete Sherany

Auch hier fällt wieder die fehlende Formatierung auf, die wir schon bei früheren HTML-Fragmenten bemerkt haben, da hierauf noch keine CSS-Regeln angewendet worden sind.

Da wir hier zeigen wollen, wie Daten an den Server übergeben werden, verwenden wir hier eine andere Methode, um Einträge anzufordern, als einzelne Schaltflächen, auf die wir uns bis jetzt verlassen haben. Stattdessen zeigen wir eine Liste von Links für jeden Begriff an, wobei ein Klick auf einen davon zu der jeweiligen Definition führt. Der HTML-Code dafür sieht wie folgt aus:

```
<div class="letter" id="letter-e">
<h3>E</h3>
<ul>
  <li><a href="e.php?term=Eavesdrop">Eavesdrop</a></li>
  <li><a href="e.php?term=Edible">Edible</a></li>
  <li><a href="e.php?term=Education">Education</a></li>
  <li><a href="e.php?term=Eloquence">Eloquence</a></li>
  <li><a href="e.php?term=Elysium">Elysium</a></li>
  <li><a href="e.php?term=Emancipation">Emancipation</a></li>
  <li><a href="e.php?term=Emotion">Emotion</a></li>
  <li><a href="e.php?term=Envelope">Envelope</a></li>
  <li><a href="e.php?term=Envy">Envy</a></li>
  <li><a href="e.php?term=Epitaph">Epitaph</a></li>
  <li><a href="e.php?term=Evangelist">Evangelist</a></li>
</ul>
</div>
```

Jetzt müssen wir unseren JavaScript-Code noch dazu bringen, das PHP-Skript mit den richtigen Parametern aufzurufen. Dazu könnten wir den normalen `.load()`-Mechanismus verwenden und dabei den Abfragestring direkt an die URL anhängen, sodass wir die Daten mit Adressen wie `e.php?term=eavesdrop` abrufen.

Stattdessen lassen wir den Abfragestring jedoch von jQuery auf Basis der Zuordnung konstruieren, die wir der Funktion `$.get()` übergeben:

```
$(document).ready(function() {
    $('#letter-e a').click(function() {
        var requestData = {term: $(this).text()};
        $.get('e.php', requestData, function(data) {
            $('#dictionary').html(data);
        });
        return false;
    });
});
```

**Listing 6-10**

Da wir bereits andere Ajax-Schnittstellen gesehen haben, die jQuery zur Verfügung stellt, kommt uns die Operation dieser Funktion vertraut vor. Der einzige Unterschied ist der zweite Parameter, der es uns ermöglicht, eine Zuordnung von Schlüsseln und Werten bereitzustellen, die in den Abfragestring eingebaut wird. In diesem Fall ist der Schlüssel immer der `term`, während der Wert aus dem Text des jeweiligen Links entnommen wird. Wenn Sie jetzt auf den ersten Link in der Liste klicken, wird die zugehörige Definition angezeigt:

The screenshot shows a web page titled "The Devil's Dictionary" by Ambrose Bierce. The page lists words starting with 'A', 'B', 'C', 'D', and 'E'. The word 'EAVESDROP' is highlighted in blue. Its definition is: "Secretly to overhear a catalogue of the crimes and vices of another or yourself." Below the definition, there is a short story. To the right of the story, under 'E', are the words "Gopete" and "Sherany". At the bottom left, there is a list of links: Eavesdrop, Edible, Education, Eloquence, Elysium, Emancipation, Emotion.

Bei all diesen Links sind Adressen angegeben, auch wenn wir sie in diesem Code nicht verwenden. Damit bieten wir Benutzern, die JavaScript deaktiviert oder aus anderen Gründen nicht zur Verfügung haben, eine alternative Möglichkeit, um in den Informationen zu navigieren (eine Form von *fortschreitender Verbesserung*). Damit die Links beim Anklicken nicht auf die normale Weise funktionieren, gibt der Ereignishandler `false` zurück.

### 6.3.2 POST-Requests durchführen

HTTP-Requests mit der Methode POST sind fast identisch mit der GET-Variante. Einer der deutlichsten Unterschiede besteht darin, dass GET sein Argument in den Teil der URL mit dem Abfragestring stellt, was bei POST nicht geschieht. Bei Ajax-Aufrufen ist aber selbst dieser Unterschied für den normalen Benutzer nicht zu erkennen. Die einzigen Gründe, um eine dieser Methoden der anderen vorzuziehen, bestehen im Allgemeinen darin, den Normen des serverseitigen Codes zu genügen oder große Mengen übertragener Daten bereitzustellen, denn GET weist strengere Einschränkungen auf. Wir haben den Code für unser PHP-Beispiel so geschrieben, dass sich beide Methoden gleich gut eignen, weshalb wir einfach von GET zu POST wechseln können, indem wir die aufzurufende jQuery-Funktion ändern:

```
$(document).ready(function() {
    $('#letter-e a').click(function() {
        var requestData = {term: $(this).text()};
        $.post('e.php', requestData, function(data) {
            $('#dictionary').html(data);
        });
        return false;
    });
});
```

**Listing 6-11**

Die Argumente sind dieselben, doch die Anforderung erfolgt jetzt über POST. Diesen Code können wir noch weiter vereinfachen, indem wir die Methode .load() einsetzen, die standardmäßig POST verwendet, wenn sie mit einer Zuordnung von Argumenten versehen wird:

```
$(document).ready(function() {
    $('#letter-e a').click(function() {
        var requestData = {term: $(this).text()};
        $('#dictionary').load('e.php', requestData);
        return false;
    });
});
```

**Listing 6-12**

Diese abgespeckte Version funktioniert bei einem Klick auf einen Link genauso wie die andere, wie der folgende Screenshot zeigt:

The screenshot shows a page from 'The Devil's Dictionary' by Ambrose Bierce. The title 'The Devil's Dictionary' is at the top, followed by 'by Ambrose Bierce'. Below this is a table of contents with entries A through E. The entry for 'E' is expanded, showing its definition and several related words. The right side of the table of contents lists some of these related words with their definitions.

A	EAVESDROP v.i. Secretly to overhear a catalogue of the crimes and vices of another or yourself.
B	
C	A lady with one of her ears applied To an open keyhole heard, inside, Two female gossips in converse free — The subject engaging them was she.
D	"I think," said one, "and my husband thinks That she's a prying, inquisitive minx!"
E	As soon as no more of it she could hear The lady, Indignant, removed her ear. "I will not stay," she said, with a pout, "To hear my character lied about!"
Eavesdrop	Gopete
Edible	Sherany
Education	
Eloquence	
Elysium	
Emancipation	
Emotion	

### 6.3.3 Formulare serialisieren

Wenn Daten an einen Server gesendet werden sollen, muss der Benutzer häufig ein Formular ausfüllen. Anstatt uns auf den normalen Mechanismus zur Formularübertragung zu verlassen, der die Antwort in das gesamte Browserfenster lädt, können wir den Ajax-Werkzeugkasten von jQuery nutzen, um das Formular asynchron zu übertragen und die Antwort dann innerhalb der aktuellen Seite zu platzieren.

Um dies auszuprobieren, müssen wir ein einfaches Formular erstellen:

```
<div class="letter" id="letter-f">
    <h3>F</h3>
    <form action="f.php">
        <input type="text" name="term" value="" id="term" />
        <input type="submit" name="search" value="search"
               id="search" />
    </form>
</div>
```

Hier erhalten wir vom Server eine Menge von Einträgen zurück, indem wir mit Hilfe von PHP nach dem angegebenen Suchbegriff forschen, der ein Teilstring des Wörterbuchbegriffs ist. Wir verwenden die Funktion `build_entry()` aus `e.php`, um die Daten im selben Format zurückzugeben wie zuvor, ändern aber die Logik in `f.php`:

```
<?php
$output = array();
foreach ($entries as $term => $entry) {
    if (strpos($term, strtoupper($_REQUEST['term'])) !== FALSE) {
        $output[] = build_entry($term, $entry);
    }
}
```

```
if (!empty($output)) {
    echo implode("\n", $output);
} else {
    echo '<div class="entry">Sorry, no entries found for ';
    echo '<strong>' . $_REQUEST['term'] . '</strong>.';
    echo '</div>';
}
?>
```

Der Aufruf von `strpos()` durchsucht das Wort nach dem angegebenen Suchstring. Jetzt können wir auf die Formularvorlage reagieren und die entsprechenden Abfrageparameter erstellen, indem wir den DOM-Baum durchlaufen:

```
$(document).ready(function() {
    $('#letter-f form').submit(function(event) {
        event.preventDefault();
        $.get('f.php', {'term': $('input[name="term"]').val()},
            function(data) {
                $('#dictionary').html(data);
            });
    });
});
```

**Listing 6–13**

#### Standardaktion verhindern oder "false" zurückgeben?

In Listing 6–13 haben wir das `event`-Objekt im `submit`-Handler übergeben und `event.preventDefault()` verwendet, anstatt den Handler mit `return false` abzuschließen. Diese Vorgehensweise wird empfohlen, wenn die Standardaktion sonst dazu führen würde, dass eine andere Seite oder erneut die aktuelle Seite geladen würde. Wenn unser `submit`-Handler beispielsweise einen JavaScript-Fehler enthält, dann sorgt die Verhinderung der Standardaktion in der ersten Zeile des Handlers dafür, dass das Formular nicht übertragen wird, sodass die Fehlerkonsole des Browsers den Fehler ordnungsgemäß meldet. In Kapitel 3, »Ereignisbehandlung«, haben wir jedoch gelernt, dass `return false` sowohl `event.preventDefault()` als auch `event.stopPropagation()` aufruft. Um ein Event Bubbling zu verhindern, müssen wir auch letztere Methode aufrufen.

Dieser Code zeigt die gewünschte Wirkung, aber Eingabefelder anhand ihres Namens zu suchen und an eine Zuordnung anzuhängen, ist mühselig, vor allem, wenn das Formular komplizierter wird. Glücklicherweise bietet jQuery eine Abkürzung für diesen häufig genutzten Ausdruck. Die Methode `.serialize()` verarbeitet ein jQuery-Objekt und übersetzt die ausgewählten DOM-Elemente in einen Abfragestring, der zusammen mit einer Ajax-Anforderung übergeben werden kann. Unseren Übertragungshandler können wir damit wie folgt verallgemeinern:

```
$(document).ready(function() {
    $('#letter-f form').submit(function(event) {
        event.preventDefault();
        var formValues = $(this).serialize();
        $.get('f.php', formValues, function(data) {
            $('#dictionary').html(data);
        });
    });
});
```

**Listing 6–14**

Jetzt funktioniert das Skript zur Übertragung des Formulars auch dann, wenn sich die Anzahl der Felder erhöht. Wenn wir beispielsweise eine Suche nach fid durchführen, werden die Begriffe angezeigt, die diesen Teilstring enthalten:

The Devil's Dictionary  
by Ambrose Bierce

**A** **FIDDLE** *n.*  
An instrument to tickle human ears by friction of a horse's tail on the entrails of a cat.  
**B** To Rome said Nero: "If to smoke you turn I shall not cease to fiddle while you burn."  
**C** To Nero Rome replied: "Pray do your worst,"  
**D** 'Tis my excuse that you were fiddling first.  
**E**  
Eavesdrop **FIDELITY** *n.*  
Edible A virtue peculiar to those who are about to be betrayed.  
Education  
Flimflam

## 6.4 Unterschiedliche Inhalte liefern

Wir haben bereits bei der Übertragung von HTML-Daten gezeigt, dass die Dokumentfragmente unformatiert erscheinen, wenn wir sie im Browser aufrufen, anstatt JavaScript zu verwenden. Um Benutzern ohne JavaScript ein schöneres Erscheinungsbild zu bieten, können wir in diesem Fall ein komplettes Dokument mit <html>-, <head>- und <body>-Elementen und all ihren Inhalten laden. Dazu lässt sich der Request-Header nutzen, den jQuery mit jeder Ajax-Anforderung sendet. In unserem serverseitigen Code (in diesem Fall ist es PHP-Code) müssen wir nur nach dem Header X-Requested-With Ausschau halten. Wenn er gesetzt ist und den Wert XMLHttpRequest aufweist, stellen wir nur das Fragment bereit, andernfalls liefern wir das gesamte Dokument. Eine grundlegende Implementierung in PHP sieht wie folgt aus:

```
<?php
$ajax = isset($_SERVER['HTTP_X_REQUESTED_WITH']) &&
$_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest';

if (!$ajax):
// Zeigt <head> und den Anfang von <body> für Nicht-Ajax-Anforderungen an
?>
<!DOCTYPE HTML>
<html lang="en">
<head>
    <!-- Titel, Metainformationen, Links -->
</head>
<body>
    <!-- Seitenüberschrift, Formulare usw. -->
<?php
endif;

// Zeigt Eintragsinformationen sowohl für Ajax als auch für Nicht-Ajax an

if (!$ajax):
// Schließt offene <div>s, <body> und <html> für Nicht-Ajax
?>

</body>
</html>

<?php endif; ?>
```

Damit haben wir jetzt ein echtes Beispiel für *fortschreitende Verbesserung*, bei dem Benutzer ohne JavaScript ein verwendbares Formular mit formatierten Ergebnissen sehen, während diejenigen mit JavaScript ein verbessertes Erscheinungsbild genießen können.

Mit einer solchen Einrichtung in einem Serverskript lassen sich sogar Daten mit viel gravierenderen Unterschieden zurückgeben. Beispielsweise können wir für Ajax-Anforderungen JSON-Daten zurückgeben und für andere Anfragen HTML-Daten:

```
<?php
$ajax = isset($_SERVER['HTTP_X_REQUESTED_WITH']) &&
$_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest';

// Richtet das Array $entries ein

if ($ajax) {
header('Content-type: application/json');
echo json_encode($entries);
}
else {
    // Gibt das vollständige HTML-Dokument aus
}
```

Das führt zwar dazu, dass weniger Daten zu übertragen sind, aber nach dem Empfang der Daten muss die HTML-Struktur aufgebaut werden, wie wir es in Listing 6–9 getan haben.

## 6.5 Die Anforderung im Auge behalten

Bis jetzt haben wir uns damit begnügt, eine Ajax-Methode aufzurufen und geduldig auf die Antwort zu warten. Manchmal jedoch ist es nützlich, mehr über den Fortschritt des HTTP-Requests in Erfahrung zu bringen. Für diesen Fall bietet jQuery eine Reihe von Funktionen, mit denen sich beim Auftreten verschiedener Ajax-Ereignisse *Callbacks* registrieren lassen.

Die Methoden `.ajaxStart()` und `.ajaxStop()`, die sich an ein beliebiges jQuery-Objekt anhängen lassen, sind zwei Beispiele für solche Beobachterfunktionen. Wenn ein Ajax-Aufruf beginnt und dabei keine andere Übertragung läuft, wird der `.ajaxStart()`-Callback ausgelöst, und wenn die letzte aktive Anforderung endet, wird der mit `.ajaxStop()` verbundene Callback ausgeführt. All diese Beobachter sind *global*, d.h., sie werden aufgerufen, wenn irgendeine Form von Ajax-Kommunikation erfolgt, unabhängig davon, welcher Code sie ausgelöst hat.

Mit diesen Methoden können wir dem Benutzer bei einer langsamen Netzwerkverbindung eine gewisse Rückmeldung geben. Dem HTML-Code für die Seite können wir eine geeignete Ladenachricht anhängen:

```
<div id="loading">  
    Loading...  
</div>
```

Diese Nachricht ist einfach nur ein beliebiger HTML-Text. Wir könnten hier z.B. auch ein animiertes GIF mit einer Ladeanzeige einschließen. In diesem Beispiel fügen wir noch einige einfache Formate zu der CSS-Datei hinzu, damit die Seite bei der Anzeige der Meldung wie folgt aussieht:

The screenshot shows a portion of the 'The Devil's Dictionary' page by Ambrose Bierce. On the left, there is a vertical list of letters A through E. Below this list, there are two blue hyperlinks: 'Eavesdrop' and 'Edible'. To the right of the list, there is a light gray rectangular box containing the text 'Loading...'. The rest of the page is mostly blank white space.

Um dem Prinzip der fortschreitenden Verbesserung zu folgen, platzieren wir dieses HTML-Markup aber nicht direkt auf der Seite. Da es nur dann von Bedeutung ist, wenn JavaScript zur Verfügung steht, fügen wir es wie folgt mithilfe von jQuery ein:

```
$(document).ready(function() {
    $('<div id="loading">Loading...</div>')
        .insertBefore('#dictionary')
});
```

Unsere CSS-Datei gibt diesem <div> die Formatdeklaration `display: none`, sodass die Meldung ursprünglich verborgen ist. Um sie zum richtigen Zeitpunkt einzublenden, registrieren wir sie als Beobachter für `.ajaxStart()`:

```
$(document).ready(function() {
    $('<div id="loading">Loading...</div>')
        .insertBefore('#dictionary')
        .ajaxStart(function() {
            $(this).show();
        });
});
```

Das Ausblendverhalten können wir gleich damit verketten:

```
$(document).ready(function() {
    $('<div id="loading">Loading...</div>')
        .insertBefore('#dictionary')
        .ajaxStart(function() {
            $(this).show();
        })
        .ajaxStop(function() {
            $(this).hide();
        });
});
```

***Listing 6–15***

Voilà! Da ist sie, unsere Rückmeldung über den Ladevorgang!

Beachten Sie wiederum, dass diese Methode unabhängig davon ist, auf welche Weise die Ajax-Kommunikation eingeleitet wurde. Sowohl die mit dem Link A verknüpfte Methode `.load()` als auch die Methode `.getJSON()` für den Link B rufen diese Aktionen hervor.

In diesem Beispiel ist dieses globale Verhalten erwünscht. Wenn wir jedoch spezifischer vorgehen müssen, stehen uns mehrere Möglichkeiten zur Verfügung. Einige der Beobachtermethoden, beispielsweise `.ajaxError()`, senden in ihrem Callback einen Verweis auf das Objekt `XMLHttpRequest`. Damit lassen sich Anforderungen voneinander unterscheiden, sodass unterschiedliche Verhaltensweisen möglich sind. Eine spezifische Behandlung lässt sich auch durch die Verwendung der grundlegenderen Funktion `$.ajax()` erreichen, die wir uns etwas später ansehen werden.

Die häufigste Art der Interaktion mit der Anforderung bildet jedoch der success-Callback, den wir bereits in mehreren Beispielen verwendet haben, um die vom Server zurückkommenden Daten zu interpretieren und die Seite mit Ergebnissen zu füllen. Natürlich kann er auch für andere Arten der Rückmeldung eingesetzt werden. Betrachten Sie noch einmal unser .load()-Beispiel aus Listing 6-1:

```
$(document).ready(function() {
    $('#letter-a a').click(function() {
        $('#dictionary').load('a.html');
        return false;
    });
});
```

Hier können wir eine kleine Verbesserung vornehmen, indem wir den geladenen Inhalt langsam einblenden, anstatt ihn plötzlich erscheinen zu lassen. Die Methode .load() kann einen Callback übernehmen, der bei ihrem Abschluss ausgelöst wird:

```
$(document).ready(function() {
    $('#letter-a a').click(function() {
        $('#dictionary').hide().load('a.html', function() {
            $(this).fadeIn();
        });
        return false;
    });
});
```

#### ***Listing 6-16***

Als Erstes verbergen wir das Zielelement, dann lösen wir den Ladevorgang aus. Wenn er abgeschlossen ist, setzen wir den Callback ein, um das neue Element einzublenden.

## **6.6 Fehlerbehandlung**

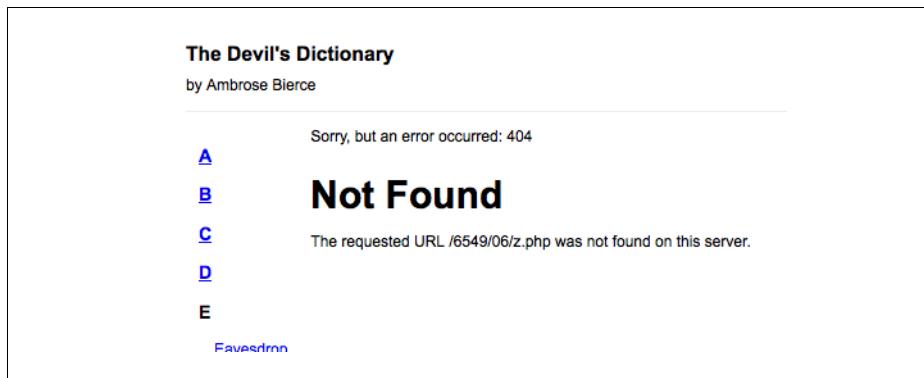
Bis jetzt haben wir uns nur mit erfolgreichen Antworten auf Ajax-Anforderungen beschäftigt, bei denen das Laden der Seiten mit neuen Inhalten wie geplant abläuft. Verantwortungsvolle Entwickler sollten jedoch die Möglichkeit von Netzwerk- und Datenfehlern einplanen und sie entsprechend melden. Die Entwicklung von Ajax-Anwendungen in einer lokalen Umgebung kann Entwickler nachlässig machen, da abgesehen von falsch eingegebenen URLs einfach keine Ajax-Fehler auftreten. Leider verfügen die Ajax-Komfortmethoden wie \$.get() und .load() nicht über eigene Argumente für Fehler-Callbacks, sodass – zumindest in den ersten Versionen von jQuery – die globale Methode .ajaxError() die einzige Möglichkeit war, Fehler bei diesen Methoden zu handhaben. Seit der Überarbeitung der Ajax-Komponente in jQuery 1.5 können wir jetzt success()-, complete()- und error()-Callbacks mit allen anderen Ajax-Funktionen außer .load() verketteten.

Beispielsweise können wir im Code aus Listing 6–16 die URL in eine Adresse ändern, die es gar nicht gibt, und dann den error()-Callback testen:

```
$(document).ready(function() {
    $('#letter-e a').click(function() {
        var requestData = {term: $(this).text()};
        $.get('z.php', requestData, function(data) {
            $('#dictionary').html(data);
        }).error(function(jqXHR) {
            $('#dictionary')
                .html('An error occurred: ' + jqXHR.status)
                .append(jqXHR.responseText);
        });
        return false;
    });
});
```

**Listing 6–17**

Wenn Sie jetzt auf einen der Links für Begriffe klicken, die mit E beginnen, wird wie im folgenden Screenshot eine Fehlermeldung angezeigt. Der genaue Inhalt von jqXHR.responseText hängt von der Serverkonfiguration ab.



Die Eigenschaft `.status` enthält einen vom Server bereitgestellten numerischen Code. Diese Codes sind in der HTTP-Spezifikation definiert und stehen für folgende Fehlerbedingungen:

Antwortcode	Beschreibung
400	Fehlerhafte Anforderung
401	Nicht autorisiert
403	Verboten
404	Nicht gefunden
500	Interner Serverfehler

Eine vollständige Liste der Antwortcodes finden Sie auf der Website des W3C unter <http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>.

Die Fehlerbehandlung sehen wir uns ausführlicher in Kapitel 13, »Ajax für Fortgeschrittene«, an.

## 6.7 Ereignisse in Ajax

Wir wollen dafür sorgen, dass die einzelnen Wörterbucheinträge jeweils die Anzeige der zugehörigen Definitionen steuern, d.h., dass bei einem Klick auf den Begriff die Definition angezeigt oder verborgen werden soll. Mit den bisher vorgestellten Techniken ist das nicht weiter schwierig:

```
// Unfinished code
$(document).ready(function() {
    $('h3.term').click(function() {
        $(this).siblings('.definition').slideToggle();
    });
});
```

**Listing 6-18**

Beim Klick auf einen Begriff findet der Code Geschwister des Elements, die die Klasse `definition` aufweisen, und faltet sie auseinander bzw. zusammen.

Hier scheint alles in Ordnung zu sein, doch in Wirklichkeit bewirkt ein Klick gar nichts, denn leider sind die Begriffe dem Dokument noch gar nicht hinzugefügt worden, wenn wir die `click`-Handler anbringen. Selbst wenn wir es schaffen sollten, diese Handler mit den Elementen zu verknüpfen, würde diese Verbindung nach dem ersten Klick auf einen anderen Buchstaben wieder gelöst werden.

Das ist ein häufiges Problem bei Seitenbereichen, die von Ajax ausgefüllt werden. Eine verbreitete Lösung besteht darin, die Handler bei jeder Aktualisierung des Seitenbereichs neu zu binden. Das kann jedoch mühselig sein, da der Code für die Ereignisbindung jedes Mal aufgerufen werden muss, wenn irgend etwas die DOM-Struktur der Seite ändert.

In Kapitel 3 haben wir eine zumeist bessere Alternative kennengelernt: Wir können eine *Ereignisdelegierung* einsetzen, indem wir das Ereignis an ein Vorfahrenelement binden, das sich niemals ändert. In diesem Fall verknüpfen wir den `click`-Handler mithilfe von `.live()` mit dem Dokument und erfassen die Klicks auf diese Weise:

```
$(document).ready(function() {
    $('h3.term').live('click', function() {
        $(this).siblings('.definition').slideToggle();
    });
});
```

**Listing 6-19**

Die Methode `.live()` weist den Browser an, alle Klicks zu beobachten, die irgendwo im Dokument stattfinden. Der Handler wird genau dann ausgeführt, wenn das angeklickte Element mit dem Selektor `h3.term` übereinstimmt. Jetzt funktioniert das Umschaltverhalten bei jedem Begriff, auch wenn er erst durch eine spätere Ajax-Transaktion hinzugefügt wird.

## **6.8 Sicherheitseinschränkungen**

Trotz allem Nutzen beim Erstellen dynamischer Webanwendungen unterliegt XMLHttpRequest (die zugrunde liegende Browsetechnologie hinter der Ajax-Implementierung von jQuery) strengen Einschränkungen. Um verschiedene Arten von *Cross-Site-Scripting-Angriffen* zu verhindern, ist es im Allgemeinen nicht möglich, ein Dokument von einem anderen Server als dem anzufordern, auf dem die ursprüngliche Seite ausgeführt wird.

Das ist meistens eine gute Sache. Beispielsweise wird die Implementierung der JSON-Analyse mit `eval()` als unsicher bezeichnet. Wenn in der Datendatei schädlicher Code vorhanden ist, könnte er durch den Aufruf von `eval()` ausgeführt werden. Da sich die Datendatei jedoch auf demselben Server befinden muss wie die Webseite selbst, ist die Wahrscheinlichkeit, Code in die Datendatei einzuschleusen, ungefähr genauso hoch wie die, Code direkt auf die Webseite einzuschmuggeln. Das bedeutet, dass `eval()` beim Laden vertrauenswürdiger JSON-Dateien kein ernstes Sicherheitsproblem darstellt.

Wenn jQuery JSON-Daten analysiert, vermeidet es die Verwendung von `eval()`, sondern versucht es zunächst mit der browsereigenen Methode `JSON.parse()`. Falls der Browser diese Methode nicht unterstützt, wird der Konstruktor `Function` verwendet, um die Daten auszuwerten, wobei das Ergebnis von `(new Function("return" + data))()` zurückgegeben wird. Diese Analyse mit Test der unterstützten Funktionen ist in der Funktion `jQuery.parseJSON()` abstrahiert.

Es gibt jedoch viele Fälle, in denen es vorteilhaft wäre, Daten von einer anderen Quelle laden zu können. Um das zu erreichen und die Sicherheitseinschränkungen zu umgehen, gibt es eine Reihe von Möglichkeiten.

Eine Methode besteht darin, es dem Server zu überlassen, die fremden Daten zu laden und bei Anforderung durch den Client bereitzustellen. Das ist ein sehr leistungsfähiger Ansatz, da der Server die Daten bei Bedarf auch einer Vorverarbeitung unterziehen kann. Beispielsweise könnten wir XML-Daten mit RSS-Newssfeeds aus verschiedenen Quellen laden und auf dem Server zu einem einzigen Feed zusammenfassen, den wir dann auf Anforderung dem Client zur Verfügung stellen.

Um Daten ohne Einbeziehung des Servers von einer anderen Stelle zu laden, müssen wir jedoch raffinierter vorgehen. Ein beliebter Ansatz zum Laden fremder JavaScript-Dateien besteht darin, `<script>`-Tags bei Bedarf einzubauen. Da jQuery uns dabei hilft, neue DOM-Elemente einzufügen, ist das so einfach wie im folgenden Beispiel:

```
$(document.createElement('script'))
    .attr('src', 'http://example.com/example.js')
    .appendTo('head');
```

Die Methode `$.getScript()` passt sich sogar automatisch dieser Technik an, wenn sie in ihrem URL-Argument einen fremden Host findet, sodass uns diese Aufgabe abgenommen wird.

Der Browser führt das geladene Skript zwar aus, allerdings gibt es keinen Mechanismus, um die Ergebnisse dieses Skripts abzurufen. Daher erfordert diese Technik ein wenig Mitarbeit von dem fremden Host. Das geladene Skript muss irgendeine Aktion durchführen, z.B. eine globale Variable setzen, die eine Auswirkung auf die lokale Umgebung hat. Dienste, die auf diese Weise ausführbare Skripte veröffentlichen, bieten auch eine API an, über die wir mit dem fremden Skript in Interaktion treten können.

Eine weitere Möglichkeit besteht darin, das HTML-Tag `<iframe>` zu nutzen, um die fremden Daten zu laden. Dieses Element erlaubt es, jede URL als Quelle für den Datenabruf zu verwenden, selbst wenn sie nicht mit dem Server der zugehörigen Seite übereinstimmt. Die Daten können geladen und auf der aktuellen Seite angezeigt werden. Um sie bearbeiten zu können, ist jedoch dieselbe Mitarbeit erforderlich wie für die Vorgehensweise mit dem `<script>`-Tag. Skripte innerhalb von `<iframe>` müssen die Daten ausdrücklich den Objekten im Elterndokument bereitstellen.

### Cross-Origin Resource Sharing

Eine neuere Technik namens *CORS (Cross-Origin Resource Sharing)* liegt inzwischen als Entwurf für eine W3C-Spezifikation vor. Dazu sendet die eine Domäne der anderen einen HTTP-Header, den diese erwartet. Die Empfängerdomäne muss dann den Antwort-Header `Access-Control-Allow-Origin` zurücksenden, in dem sie mitteilt, dass die Domäne akzeptiert ist. Weitere Informationen über CORS finden Sie unter <http://w3.org/TR/cors/>.

#### 6.8.1 JSONP für fremde Daten verwenden

Die Methode `<script>`-Tags zum Abrufen von JavaScript-Dateien von einer fremden Quelle kann variiert werden, um auch JSON-Dateien von einem anderen Server zu holen. Dazu müssen wir die JSON-Datei auf dem Server jedoch leicht verändern. Dafür gibt es mehrere Mechanismen, von denen einer direkt von jQuery unterstützt wird, nämlich *JSONP (JSON with Padding)*.

Das Dateiformat JSONP besteht aus einer standardmäßigen JSON-Datei, die in Klammern eingeschlossen und mit einem vorausgehenden beliebigen Textstring versehen ist. Dieser String, die »Auffüllung« (*padding*), wird von dem Client bestimmt, der die Daten anfordert. Aufgrund der Klammern kann der Client je nachdem, was als Auffüllstring gesendet wird, dafür sorgen, dass eine Funktion aufgerufen oder dass eine Variable gesetzt wird.

Eine PHP-Implementierung der JSONP-Technik ist ziemlich einfach:

```
<?php  
    print($_GET['callback'] .('.$data.'));  
?>
```

Hier ist \$data eine Variable mit einer Stringdarstellung einer JSON-Datei. Beim Aufruf des Skripts wird der Abfragestringparameter callback vorn an die resultierende Datei angehängt, die zum Client zurückgeht.

Um diese Technik vorzuführen, müssen wir unser früheres JSON-Beispiel aus Listing 6–6 nur leicht verändern, damit es diese fremde Datenquelle aufruft. Die Funktion \$.getJSON() nutzt dazu das besondere Platzhalterzeichen ?, wie Sie im folgenden Code sehen:

```
$(document).ready(function() {  
    var url = 'http://examples.learningjquery.com/jsonp/g.php';  
    $('#letter-g a').click(function() {  
        $.getJSON(url + '?callback=?', function(data) {  
            var html = '';  
            $.each(data, function(entryIndex, entry) {  
                html += '<div class="entry">';  
                html += '<h3 class="term">' + entry.term + '</h3>';  
                html += '<div class="part">' + entry.part + '</div>';  
                html += '<div class="definition">';  
                html += entry.definition;  
                if (entry.quote) {  
                    html += '<div class="quote">';  
                    $.each(entry.quote, function(lineIndex, line) {  
                        html += '<div class="quote-line">' + line +  
                            '</div>';  
                    });  
                    if (entry.author) {  
                        html += '<div class="quote-author">' +  
                            entry.author + '</div>';  
                    }  
                    html += '</div>';  
                }  
                html += '</div>';  
                html += '</div>';  
            });  
            $('#dictionary').html(html);  
        });  
    });  
});
```

```
        return false;
    });
});
});
```

**Listing 6–20**

Normalerweise dürften wir keine JSON-Daten von einem anderen Server abrufen (in diesem Beispiel von `examples.learningjquery.com`). Da die Datei jedoch so einrichtet ist, dass sie ihre Daten im JSONP-Format bereitstellt, können wir die Daten bekommen, indem wir den Abfragestring an unsere URL anhängen und dabei `?` als Platzhalter für den Wert des `callback`-Arguments verwenden. Wenn diese Anforderung erfolgt, ersetzt jQuery das `?`, analysiert das Ergebnis und über gibt es als `data` an die Funktion `success`, als ob es sich um eine lokale JSON-Anforderung handeln würde.

Beachten Sie, dass hier immer noch dieselben Sicherheitsbedenken gelten. Was immer der Server dem Browser zurückgibt, wird auf dem Computer des Benutzers ausgeführt. Die JSONP-Technik sollte nur für Daten aus vertrauenswürdigen Quellen eingesetzt werden.

## 6.9 Zusätzliche Optionen

Der Ajax-Werkzeugkasten von jQuery ist reich bestückt. Wir haben uns bereits verschiedene der verfügbaren Optionen angesehen, dabei jedoch nur an der Oberfläche gekratzt. Da es zu viele Varianten gibt, um alle hier zu behandeln, geben wir stattdessen einen Überblick über einige der herausragenden Möglichkeiten, die Ajax-Kommunikation anzupassen.

### 6.9.1 Die grundlegende Methode ajax

Wir haben uns bereits verschiedene Methoden angesehen, die Ajax-Transaktionen auslösen. Intern ordnet jQuery all diese Methoden zu Varianten der globalen Funktion `$.ajax()` zu. Anstatt eine bestimmte Art von Ajax-Aktivität vorauszusetzen, nimmt diese Funktion eine Zuordnung der Optionen entgegen, mit denen ihr Verhalten angepasst werden kann.

Unser erstes Beispiel, Listing 6–1, hat ein HTML-Fragment mit `$('#dictionary').load('a.html')` geladen. Diese Aktion hätten wir auch wie folgt mit `$.ajax()` durchführen können:

```
$.ajax({
  url: 'a.html',
  success: function(data) {
    $('#dictionary').html(data);
  }
});
```

**Listing 6–21**

Statt einzelner Argumente für die URL, die Daten und einen Erfolgs-Callback übernimmt `$.ajax` eine einzige Zuordnung für über 30 Einstellungen, was sehr viel Flexibilität bietet. Im Folgenden sehen Sie eine kleine Auswahl der speziellen Möglichkeiten, die Sie bei einem solchen Aufruf von `$.ajax()` haben:

- Verhindern, dass der Browser die Antwort vom Server zwischenspeichert. Das kann sinnvoll sein, wenn der Server die Daten dynamisch liefert.
- Registrieren getrennter Callback-Funktionen für den erfolgreichen Abschluss der Anforderung, für den Abschluss mit Fehler oder für beide Arten.
- Unterdrücken der globalen Handler (z.B. diejenigen, die für `$.ajaxStart()` registriert werden), die normalerweise von allen Ajax-Interaktionen ausgelöst werden.
- Bereitstellen eines Benutzernamens und eines Kennworts für die Authentifizierung beim fremden Host.

Einzelheiten über diese und andere Optionen erfahren Sie in Anhang C, im *jQuery Reference Guide* und in der API-Onlinerefenz (<http://api.jquery.com/jQuery.ajax>).

### **6.9.2 Standardoptionen ändern**

Mit der Funktion `$.ajaxSetup()` können wir die Standardwerte für jede der verwendeten Optionen beim Aufruf von Ajax-Methoden ändern. Diese Funktion nimmt eine Zuordnung von Optionen entgegen, die identisch mit der für `$.ajax()` ist, und sorgt dafür, dass diese Werte bei allen nachfolgenden Ajax-Anforderungen zur Anwendung kommen, sofern sie nicht überschrieben werden:

```
$.ajaxSetup({  
    url: 'a.html',  
    type: 'POST',  
    dataType: 'html'  
});  
  
$.ajax({  
    type: 'GET',  
    success: function(data) {  
        $('#dictionary').html(data);  
    }  
});
```

**Listing 6–22**

Die Reihenfolge der Operationen entspricht der in unserem vorangegangenen Beispiel zu `$.ajax()`. Beachten Sie, dass die URL der Anforderung durch den Aufruf von `$.ajaxSetup()` als Standardwert festgelegt wird, weshalb sie beim Aufruf von `$.ajax()` nicht angegeben werden muss. Dagegen hat der Parameter `type` den Standardwert `POST` erhalten, kann aber im `$.ajax()`-Aufruf immer noch mit `GET` überschrieben werden.

### 6.9.3 Teile einer HTML-Seite laden

Die erste und einfachste Ajax-Technik, die wir behandelt haben, bestand darin, ein HTML-Fragment abzurufen und auf einer Seite zu platzieren. Manchmal bietet der Server schon den HTML-Code, den wir brauchen, umgibt ihn aber mit einer HTML-Seite, die wir nicht haben wollen. Wir haben uns bereits angesehen, wie wir den Server so einrichten, dass er auf Anfragen unterschiedliche Inhalte zurückgeben kann. Wenn es aber zu umständlich ist, den Server die Daten in dem von uns gewünschten Format zurückgeben zu lassen, kann jQuery uns auf der Clientseite weiterhelfen.

Kehren Sie zu unserem ersten Beispiel zurück, stellen Sie sich aber dabei vor, dass das Dokument mit den Wörterbucheinträgen eine vollständige HTML-Seite namens `h.html` ist:

```
<html lang="en">
  <head>
    <meta charset="utf-8"/>
    <title>The Devil's Dictionary: H</title>
    <link rel="stylesheet" href="dictionary.css" media="screen" />
  </head>
  <body>
    <div id="container">
      <div id="header">
        <h2>The Devil's Dictionary: H</h2>
        <div class="author">by Ambrose Bierce</div>
      </div>

      <div id="dictionary">
        <div class="entry">
          <h3 class="term">HABEAS CORPUS</h3>
          <div class="part">n.</div>
          <div class="definition">
            A writ by which a man may be taken out of jail
            when confined for the wrong crime.
          </div>
        </div>

        <div class="entry">
          <h3 class="term">HABIT</h3>
          <div class="part">n.</div>
          <div class="definition">
            A shackle for the free.
          </div>
        </div>
      </div>
    </body>
  </html>
```

Mit dem bereits früher geschriebenen Code können wir das gesamte Dokument auf unsere Seite laden:

```
// Unfinished code
$(document).ready(function() {
    $('#letter-h a').click(function() {
        $('#dictionary').load('h.html');
        return false;
    });
});
```

**Listing 6–23**

Aufgrund der HTML-Teile, die wir nicht haben wollen, führt das jedoch zu einem unschönen Ergebnis:

The Devil's Dictionary  
by Ambrose Bierce

[A](#)      **The Devil's Dictionary: H**  
[B](#)      by Ambrose Bierce  
[C](#)  
[D](#)      **HABEAS CORPUS** *n.*  
[E](#)      A writ by which a man may be  
                taken out of jail when confined for  
Favortown

Um die überflüssigen Teile zu entfernen, können wir ein weiteres Merkmal der Methode `.load()` nutzen. Zusammen mit der URL des zu ladenden Dokuments können wir jQuery auch einen Selektorausdruck angeben. Ist ein solcher Ausdruck vorhanden, wird er dazu verwendet, um einen bestimmten Teil des zu ladenden Dokuments zu finden. Nur die passenden Teile des Dokuments werden dann auf der Seite eingefügt. In diesem Fall können wir diese Technik nutzen, um nur die Wörterbucheinträge aus dem Dokument zu entnehmen und einzufügen:

```
$(document).ready(function() {
    $('#letter-h a').click(function() {
        $('#dictionary').load('h.html .entry');
        return false;
    });
});
```

**Listing 6–24**

Jetzt befinden sich die unerwünschten Teile des Dokuments nicht mehr auf der Seite, wie der folgende Screenshot zeigt:

The screenshot shows a page from 'The Devil's Dictionary' by Ambrose Bierce. The title 'The Devil's Dictionary' is at the top, followed by 'by Ambrose Bierce'. Below this is a list of entries starting with 'A' and 'B'. The entry for 'HABEAS CORPUS' is shown in bold: 'HABEAS CORPUS n. A writ by which a man may be taken out of jail when confined for the wrong crime.' The entry for 'HABIT' is also shown in bold: 'HABIT n. A shackle for the free.' The entry for 'HALF' is also shown in bold: 'HALF n. One of two equal parts into which a thing may be divided, or considered as divided. In the fourteenth century a heated discussion arose among theologists and philosophers as to whether Omnipotence could not an object into three halves, and'. There are also some other words listed like 'Eavesdrop', 'Edible', 'Education', 'Eloquence', and 'Flvslum'.

## 6.10 Zusammenfassung

In diesem Kapitel haben wir gelernt, dass die von jQuery bereitgestellten Ajax-Methoden uns dabei helfen können, Daten in verschiedenen Formaten vom Server zu laden, ohne dass dafür eine Aktualisierung der Seite notwendig wird. Wir können Skripte vom Server bei Bedarf ausführen und Daten zurück an den Server senden.

Des Weiteren haben wir gelernt, wie wir mit häufig auftretenden Problemen der Techniken zum asynchronen Laden umgehen. Dazu gehört z.B., wie wir die Bindung von Handlern erhalten, nachdem ein Ladevorgang aufgetreten ist, und wie wir Daten von einem fremden Server abrufen.

Damit ist unsere Einführung in die Kernkomponenten der Bibliothek jQuery abgeschlossen. Als Nächstes sehen wir uns an, wie wir diese Funktionsmerkmale mit jQuery-Plug-ins auf einfache Weise erweitern.

### 6.10.1 Literatur

Das Thema Ajax wird ausführlicher in Kapitel 13 behandelt. Eine vollständige Liste der verfügbaren Ajax-Methoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## **6.11 Übungsaufgaben**

Um die folgenden Übungen durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Ziehen Sie beim Laden der Seite den body-Inhalt von `exercises-content.html` in den Inhaltsbereich der Seite.
2. Anstatt das gesamte Dokument auf einmal anzuzeigen, erstellen Sie »Tooltipps« für die Buchstaben in der linken Spalte, indem Sie den zum jeweiligen Buchstaben gehörenden Inhalt aus `exercises-content.html` laden, wenn der Benutzer mit dem Mauszeiger über den Buchstaben fährt.
3. Fügen Sie eine Fehlerbehandlung für den Seitenladevorgang hinzu, bei der im Inhaltsbereich eine Fehlermeldung angezeigt wird. Testen Sie den Fehlerbehandlungscode, indem Sie das Skript so ändern, dass es statt `exercises-content.html` die Datei `does-not-exist.html` anfordert.
4. Schwierig: Senden Sie beim Laden der Seite eine JSONP-Anforderung an Twitter und rufen Sie die letzten fünf Nachrichten des Benutzers `kswedberg` ab. Fügen Sie die Nachrichten in den Inhaltsbereich der Seite ein. Die URL für den Abruf dieser Nachrichten lautet: [http://twitter.com/status/user\\_timeline/kswedberg.json?count=5](http://twitter.com/status/user_timeline/kswedberg.json?count=5).



## 7 Plug-ins verwenden

In den ersten sechs Kapiteln haben wir die jQuery-Kernkomponenten kennengelernt. Dabei wurden uns viele Möglichkeiten aufgezeigt, wie mit der jQuery-Bibliothek verschiedenste Aufgaben erfüllt werden können. So leistungsfähig wie die Bibliothek auch ist, so hat die elegante Plug-in-Architektur zur Erweiterung von jQuery durch Entwickler geführt, wodurch eine noch funktionsreichere Bibliothek entstanden ist.

Die wachsende jQuery-Anhängerschaft hat Hunderte von Plug-ins entwickelt: von kleinen Hilfsmodulen bis hin zu ausgefeilten Benutzerschnittstellen. In diesem Kapitel sehen wir uns genauer an, was ein Plug-in überhaupt ist, wie Sie von anderen entwickelte Plug-ins finden und wie Sie sie in Ihre Webseiten integrieren können. Im darauf folgenden Kapitel untersuchen wir, wie wir eigene Plug-ins der jQuery-Community zur Verfügung stellen können.

### 7.1 Plug-ins finden und Unterstützung bekommen

Die jQuery-Website bietet unter <http://plugins.jquery.com/> eine große Auswahl an Plug-ins mit Funktionen wie Bewertungen von Benutzern, Versionskontrolle und Fehlerberichten. Viele der in diesem Plug-in-Repository aufgeführten Plug-ins besitzen Links zu Demos, Beispielcode und Anleitungen für die ersten Gehversuche. Viele weitere Plug-ins finden Sie in allgemeinen Repositories wie GitHub (<http://github.com>) und direkt auf den Seiten der Entwickler. Auf GitHub erhalten Sie einen guten Eindruck über die Qualität oder zumindest die Beliebtheit eines Plug-ins, indem Sie die Kommentare durchsehen und prüfen, wie viele Entwickler das Plug-in beobachten und wie viele es verwendet haben.

Wenn Sie im Plug-in-Repository, bei GitHub und auf den Entwicklerwebseiten nicht alle Antworten auf Ihre Fragen finden können, hilft Ihnen die jQuery-Community weiter. Die jQuery-Foren bieten einen speziellen Diskussionsbereich für Plug-ins unter <http://forum.jquery.com/using-jquery-plugins>. Viele der Plug-in-Autoren sind in diesen Foren unterwegs und helfen anderen gern bei auftretenden Problemen.

## 7.2 Ein Plug-in verwenden

Es ist leicht, ein jQuery-Plug-in zu verwenden. Wir benötigen lediglich den Code des Plug-ins, verlinken in unserem HTML-Code darauf und haben die erweiterten Funktionen in unserem Skript. Um das vorzuführen, benötigen wir ein Beispiel-Plug-in.

Das jQuery-Cycle-Plug-in ist für unsere Zwecke gut geeignet. Dieses von Mike Alsup geschriebene Plug-in ermöglicht es uns, eine statische Auswahl von Seitenelementen in eine interaktive Diashow zu verwandeln. Wie viele andere beliebte Plug-ins ist es sehr einfach zu verwenden, für spezielle Anwendungen jedoch sehr gut zu konfigurieren.

### 7.2.1 Das Cycle-Plug-in herunterladen und einbinden

Wir finden das Cycle-Plug-in im jQuery-Plug-in-Repository oder auf der Plug-in-Seite unter <http://www.malsup.com/jquery/cycle/>. Diese Seite enthält Download-Anweisungen, die uns zu einem Archiv führen, das Rohfassungen und komprimierte Versionen des Plug-in-Codes enthält. In unserem Beispiel verwenden wir die Datei namens `jquery.cycle.js`.

Wenn sich das Plug-in im Verzeichnis auf unserer Seite befindet, müssen wir es im `<head>`-Tag des Dokuments referenzieren und dafür sorgen, dass es hinter der jQuery-Quelldatei und vor den Aufrufen unserer Skripten steht, die das Plug-in benutzen:

```
<head>
  <meta charset="utf-8">
  <title>jQuery Book Browser</title>
  <link rel="stylesheet" href="07.css" type="text/css" />
  <script src="jquery.js"></script>
  <script src="jquery.cycle.js"></script>
  <script src="07.js"></script>
</head>
```

Schon haben wir unser erstes Plug-in installiert. Wie Sie sehen, ist es nicht schwieriger, als jQuery selbst einzurichten. Die Funktionen des Plug-ins stehen uns jetzt in unseren Skripten zur Verfügung.

### 7.2.2 Einfache Plug-in-Anwendungen

Das Cycle-Plug-in funktioniert wie alle anderen Elemente auf der Seite auch. Um es in Aktion zu zeigen, verwenden wir folgenden einfachen HTML-Code mit Text und Bildern in Listenform:

```
<ul id="books">
  <li>
    
    <div class="title">jQuery 1.4 Reference Guide</div>
    <div class="author">Karl Swedberg</div>
    <div class="author">Jonathan Chaffer</div>
  </li>
  <li>
    
    <div class="title">jQuery Plugin Development</div>
    <div class="author">Giulio Bai</div>
  </li>
  ...
</ul>
```

Mit etwas Formatierung mittels CSS werden die Bilder nacheinander angezeigt:

## Selected jQuery Books



Das Cycle-Plug-in verwandelt die Liste in eine sehenswerte animierte Diashow. Die Transformation wird über die Methode `.cycle()` im entsprechenden Container im DOM aufgerufen:

```
$(document).ready(function() {
  $('#books').cycle();
});
```

*Listing 7-1*

Die Syntax könnte nicht einfacher sein. Wie auch bei jeder anderen integrierten jQuery-Methode wenden wir `.cycle()` auf eine jQuery-Objektinstanz an, die wiederum auf die DOM-Elemente zeigt, die wir verändern wollen. Auch ohne Argumentliste erledigt `.cycle()` für uns viele Aufgaben. Die Formatierungen auf der Seite werden so modifiziert, dass nur ein Listenelement zur gleichen Zeit angezeigt wird und alle vier Sekunden das nächste Element über einen Einblendeffekt dargestellt wird:

## Selected jQuery Books



So einfach sind alle gut geschriebenen jQuery-Plug-ins zu handhaben. Ein einfacher Methodenaufruf führt zu professionellen Ergebnissen. Wie viele andere Plug-ins auch bietet Cycle eine große Anzahl von Optionen, um seine Verhaltensweise exakt den Wünschen des Entwicklers anzupassen.

### 7.2.3 Parameter an Plug-in-Methoden übergeben

Die Übergabe von Parametern an Plug-in-Methoden funktioniert nicht anders als bei normalen jQuery-Methoden. In vielen Fällen werden die Parameter als einfache Liste von Schlüssel-Wert-Paaren (wie im vorherigen Kapitel mit `$.ajax()` gezeigt) übergeben. Die Vielzahl an Optionen kann allerdings eine Herausforderung darstellen: `.cycle()` bietet mehr als fünfzig potenzielle Konfigurationsoptionen. Die Dokumentation enthält Informationen für jede einzelne Einstellung, oft auch mit detaillierten Beispielen.

Das Cycle-Plug-in ermöglicht es uns, Geschwindigkeit und Animationsart der Dias anzupassen, wodurch die Überblendungen ausgelöst werden, und über Callbacks auf abgeschlossene Animationen zu reagieren. Um einige dieser Fähigkeiten zu demonstrieren, verwenden wir drei einfache Optionen für die Methode:

```
$(document).ready(function() {  
    $('#books').cycle({  
        timeout: 2000,  
        speed: 200,  
        pause: true  
    });  
});
```

**Listing 7-2**

Die Option `timeout` gibt die Anzahl von Millisekunden an, die zwischen den Überblendungen gewartet werden soll. Im Gegensatz dazu gibt `speed` die Anzahl von Millisekunden an, die eine Überblendung dauert. Wenn `pause` auf `true` gesetzt wird, hält die Diashow an, sobald die Maus sich auf einem Dia befindet. Was besonders nützlich ist, wenn die einzelnen Dias anklickbar sind.

#### 7.2.4 Voreingestellte Parameter

Das Cycle-Plug-in bietet schon ohne Argumente einen großartigen Nutzen. Um das zu leisten, benötigt es einen Satz voreingestellter Optionen.

Ein üblicher Ansatz, den auch Cycle verfolgt, besteht darin, alle Voreinstellungen in einem einzelnen Objekt zusammenzufassen. In diesem Fall enthält `$.fn.cycle.defaults` alle Voreinstellungen. Wenn ein Plug-in seine Voreinstellungen an einer gut erkennbaren Stelle wie dieser sammelt, können wir unseren Code auf bequeme Art und Weise anpassen, wenn wir das Plug-in mehrmals aufrufen:

```
$.fn.cycle.defaults.timeout = 10000;  
$.fn.cycle.defaults.random = true;  
  
$(document).ready(function() {  
    $('#books').cycle({  
        timeout: 2000,  
        speed: 200,  
        pause: true  
    });  
});
```

**Listing 7-3**

Hier haben wir zwei Voreinstellungen gesetzt – `timeout` und `random` –, bevor wir `.cycle()` aufgerufen haben. Da wir in unserem Aufruf für `.cycle()` einen Wert für `timeout` angegeben haben, wird die Voreinstellung ignoriert. Dagegen greift der Wert für `random`, der die Dias in zufälliger Reihenfolge überblendet.

### 7.2.5 Andere Arten von Plug-ins

Plug-ins sind nicht darauf beschränkt, nur zusätzliche jQuery-Methoden bereitzustellen. Sie können die Bibliothek auf verschiedene Weise erweitern und auch das Verhalten bestehender Funktionen ändern. Plug-ins können auch die Arbeitsweise anderer Elemente in der jQuery-Bibliothek verändern. Manche bieten Ihnen neue und einfach anzuwendende Arten von Animationen oder lösen zusätzliche jQuery-Ereignisse nach Benutzereingaben aus. Das Cycle-Plug-in bietet eine solche Erweiterung über einen neuen *benutzerdefinierten Selektor*.

#### Benutzerdefinierte Selektoren

Die Diashows von Cycle können angehalten und wieder gestartet werden, indem `.cycle('pause')` und `.cycle('resume')` aufgerufen wird. Auf diese Weise können wir leicht Schaltflächen zur Steuerung der Diashow einfügen:

```
$(document).ready(function() {
    $('<div id="books-controls"></div>').insertAfter('#books');
    $('<button>Pause</button>').click(function() {
        $('#books').cycle('pause');
        return false;
    }).appendTo('#books-controls');
    $('<button>Resume</button>').click(function() {
        $('#books').cycle('resume');
        return false;
    }).appendTo('#books-controls');
});
```

**Listing 7–4**

Nehmen wir an, dass wir mit der RESUME-Schaltfläche jede angehaltene Diashow fortsetzen wollen, für den Fall, dass es mehrere gibt. Dazu müssen wir alle `<ul>`-Elemente auf der Seite finden, die pausierende Diashows darstellen, und sie alle fortsetzen. Der Cycle-Selektor `:paused` ermöglicht uns das auf einfache Weise:

```
$(document).ready(function() {
    $('<button>Resume</button>').click(function() {
        $('ul:paused').cycle('resume');
        return false;
    }).appendTo('#books-controls');
});
```

**Listing 7–5**

Wenn Cycle geladen ist, erzeugt `($('ul:paused')` ein jQuery-Objekt, das alle Diashows auf der Seite referenziert, sodass wir mit ihnen interagieren können. Selektorerweiterungen wie diese durch Plug-ins können mit den jQuery-Standardselektoren frei kombiniert werden. Es ist offensichtlich, dass jQuery mit der Wahl des richtigen Plug-ins eine Form annehmen kann, die einfach passt.

## Plug-ins mit globalen Funktionen

Viele beliebte Plug-ins bieten innerhalb des jQuery-Namensraums neue globale Funktionen. Dies gilt für alle Plug-ins, die Funktionen bereitstellen, die sich nicht auf DOM-Elemente auf der Seite beziehen, also keine guten Kandidaten für normale jQuery-Methoden sind. Beispielsweise bietet das Cookie-Plug-in (<https://github.com/carhartl/jquery-cookie>) eine Schnittstelle zum Lesen und Schreiben von Cookie-Werten einer Seite. Die Funktionalität wird durch die Funktion `$.cookie()` bereitgestellt, die individuelle Cookies setzt oder liest.

Wenn wir uns beispielsweise merken wollen, dass Benutzer auf die Pause-Schaltfläche der Diashow geklickt haben, können wir die Schaltfläche im Pausezustand belassen, für den Fall, dass sie die Seite verlassen und später wieder erneut besuchen. Nach Laden des Cookie-Plug-ins müssen Sie zum Lesen des Cookies nur noch den Namen des Cookies als einziges Argument verwenden:

```
if ( $.cookie('cyclePaused') ) {  
    $('#books').cycle('pause');  
}
```

### Listing 7–6

Hier prüfen wir, ob ein `cyclePaused`-Cookie besteht. Der Wert spielt für uns hier keine Rolle. Wenn das Cookie existiert, gibt es eine Pause. Wenn wir diese bedingte Pause in unseren Aufruf von `.cycle()` einbauen, bleibt das erste Bild der Diashow stehen, bis der Benutzer auf die RESUME-Schaltfläche klickt.

Natürlich läuft die Diashow jetzt weiter, weil wir noch keinen Cookie gesetzt haben. Das Setzen ist jedoch genauso einfach, wie einen Wert auszulesen: Wir übergeben als zweites Argument einfach einen String, wie im folgenden Codeausschnitt gezeigt:

```
$('<div id="books-controls"></div>').insertAfter('#books');  
($('<button>Pause</button>')).click(function() {  
    $('#books').cycle('pause');  
    $.cookie('cyclePaused', 'y');  
    return false;  
}).appendTo('#books-controls');  
($('<button>Resume</button>')).click(function() {  
    $('ul:paused').cycle('resume');  
    $.cookie('cyclePaused', null);  
    return false;  
}).appendTo('#books-controls');
```

### Listing 7–7

Das Cookie wird auf `y` gesetzt, wenn die PAUSE-Schaltfläche angeklickt wird, und durch Übergabe von `null` gelöscht, wenn die RESUME-Schaltfläche ausgelöst wird. In der Voreinstellung verbleibt das Cookie in der Sitzung, also bis der Tab

im Browser geschlossen wird. Zusätzlich wird das Cookie standardmäßig der Seite zugeordnet, auf der es gesetzt wurde. Um diese Voreinstellung zu ändern, können wir eine Map mit Optionen als drittes Funktionsargument übergeben. In dieser Weise ist das Cookie-Plug-in den anderen Plug-ins und sogar den jQuery-Kernfunktionen recht ähnlich.

Um das Cookie beispielsweise über die gesamte Site verfügbar und nach sieben Tagen ungültig zu machen, rufen wir die Funktion `$.cookie('cyclePaused', 'y', {path: '/', expires: 7})` auf. Weitere Informationen hierzu und zu anderen Optionen beim Aufruf von `$.cookie()` finden sich in der Dokumentation des Plug-ins.

## 7.3 Die UI-Plug-in-Bibliothek von jQuery

Während sich die meisten Plug-ins, wie Cycle und Cookie, auf eine einzelne Aufgabe beziehen, packt jQuery UI eine Reihe von Herausforderungen an. Obwohl jQuery UI als einzelnes Plug-in auftritt, ist es eher als umfangreiche Sammlung von zusammengehörigen Plug-ins zu verstehen.

Das jQuery-UI-Team hat eine Reihe von Kernkomponenten für die Interaktion und ausgefeilte Widgets zusammengestellt, die das Webinterface mehr wie die Benutzerschnittstelle am Desktop-PC erscheinen lassen. *Interaktionskomponenten* bieten Methoden, um Objekte zu ziehen und abzulegen, zu sortieren, auszuwählen und um ihre Größe zu ändern. Die Auswahl an Widgets enthält Schaltflächen, Akkordeons, Datumswähler, Dialoge, Schieber (Slider), Fortschrittsbalken und Tabs, wobei viele weitere in Entwicklung sind. Zusätzlich bietet jQuery einen umfassenden Satz erweiterter Effekte, um die jQuery-Kernanimationen zu unterstützen.

Die vollständige UI-Bibliothek ist zu umfangreich, um sie in diesem Kapitel eingehender zu behandeln. Tatsächlich gibt es zu diesem Thema ganze Bücher. Ein wesentlicher Schwerpunkt des Projekts liegt erfreulicherweise auf der Konsistenz der Funktionen, sodass wir mit einigen Beispielen in diesem Kapitel auch den Rest der Funktionen verstehen lernen.

Downloads, Dokumentationen und Demos der jQuery-UI-Module sind unter <http://jqueryui.com> zu finden. Die Download-Seite bietet sowohl einen kombinierten Download mit allen Funktionen als auch einen konfigurierbaren, der nur die gewünschten Bestandteile enthält.

### 7.3.1 Effekte

Das Effektmodul von jQuery-UI besteht aus einem Kern und einem Satz unabhängiger Effektkomponenten. Die Kerndatei bietet Animationen für Farben und Klassen sowie für anspruchsvolle Easing-Funktionen.

## Farbanimationen

Wenn die jQuery-UI-Komponente für die Kerneffekte mit dem Dokument verbunden ist, wird die Methode `.animate()` erweitert, sodass sie zusätzliche Formateigenschaften wie `borderTopColor`, `backgroundColor` und `color` akzeptiert. Beispielweise können wir nun ein Element aus weißem Text auf schwarzem Hintergrund schrittweise in schwarzen Text auf hellgrauem Hintergrund umwandeln:

```
$(document).ready(function() {
    $('#books').hover(function() {
        $('#books .title').animate({
            backgroundColor: '#eee',
            color: '#000'
        }, 1000);
    }, function() {
        $('#books .title').animate({
            backgroundColor: '#000',
            color: '#fff'
        }, 1000);
    });
});
```

**Listing 7-8**

Wenn der Mauszeiger jetzt den Bildschirmbereich mit der Diashow erreicht, werden die Schriftfarbe des Buchtitels und die Hintergrundfarbe innerhalb einer Sekunde (1000 ms) langsam animiert:



## Klassenanimationen

Die drei Klassenmethoden, mit denen wir uns in den vorhergehenden Kapiteln beschäftigt haben – `.addClass()`, `.removeClass()` und `.toggleClass()` –, werden durch jQuery-UI erweitert und akzeptieren ein zweites Argument für die Dauer der Animation. Wird diese Dauer angegeben, verhält sich die Seite so, als wenn `.animate()` aufgerufen worden wäre und alle zu verändernden Formatattribute der Klasse dem Element direkt angegeben worden wären:

```
$(document).ready(function() {  
    $('h1').click(function() {  
        $(this).toggleClass('highlighted', 'slow');  
    });  
});
```

**Listing 7-9**

Durch das Einfügen des Codes in Listing 7-9 haben wir einen Klick auf den Seiten-Header erzeugt, der die Klasse `highlighted` hinzufügt oder entfernt. Da wir die Ausführungsgeschwindigkeit jedoch mit `slow` angegeben haben, sieht die Änderung von Farbe, Rahmen und Rändern eher wie eine Animation als wie eine spontane Änderung aus.

## Selected jQuery Books

### Easing für Fortgeschrittene

Wenn wir jQuery anweisen, eine Animation über einen bestimmten Zeitraum durchzuführen, geschieht dies nicht mit einer konstanten Rate. Wenn wir beispielsweise `$('#my-div').slideUp(1000)` aufrufen, wissen wir, dass es eine ganze Sekunde dauert, bis die Elementhöhe null erreicht. Am Anfang und zum Ende ändert sich die Höhe nur langsam und in der Mitte schnell. Diese variable Änderungsrate, Easing genannt, lässt eine Animation weicher und natürlicher aussehen.

Anspruchsvolle Easing-Funktionen variieren diese Beschleunigung bzw. Verlangsamung, um für charakteristische Ergebnisse zu sorgen. Zum Beispiel wächst die Funktion `easeInExpo` exponentiell und endet in einer vielfach schnelleren Animation als zum Startzeitpunkt. Wir können eigene Easing-Funktionen in jeder jQuery-Animationsmethode oder jQuery-UI-Effektmethode angeben. Dies geschieht, indem entweder ein Argument oder eine Option angefügt wird, abhängig von der verwendeten Syntax.

Um diesen Vorgang in Aktion zu sehen, verwenden wir `easeInExpo` als Format für die Methode `.toggleClass()`, die wir gerade in Listing 7-9 eingeführt haben:

```
$(document).ready(function() {  
    $('h1').click(function() {  
        $(this).toggleClass('highlighted', 'slow', 'easeInExpo');  
    });  
});
```

**Listing 7-10**

Wann immer jetzt der Header angeklickt wird, erscheinen die Klassenattribute zunächst schrittweise, beschleunigen dann und beenden den Übergang abrupt.

### Easing-Funktionen anschauen

Eine große Auswahl an Easing-Funktionen finden Sie unter <http://jqueryui.com/demos/effect/#easing>.

### Zusätzliche Effekte

Die einzelnen in jQuery UI enthaltenen Effektdateien bieten unterschiedlichste Überblendeffekte, von denen einige etwas komplexer sind als die einfachen Ein- und Ausblendungen, die von jQuery selbst bereitgestellt werden. Diese Effekte werden über die `.effect()`-Methode aufgerufen, die von jQuery beigesteuert wird. Effekte, die ein Element verbergen oder anzeigen, können auch über `.show()`, `.hide()` und `.toggle()` aufgerufen werden.

Die von jQuery gebotenen Effekte lassen sich vielfältig einsetzen. Manche, wie `transfer` oder `size`, sind sinnvoll, wenn Elemente ihre Form und Position ändern sollen. Andere, wie `explode` und `puff`, bieten hübsche Animationen zum Verbergen. Wieder andere, z.B. `pulsate` und `shake`, lenken die Aufmerksamkeit auf ein Element.

### Effekte anschauen

Alle jQuery-UI-Effekte finden Sie unter <http://jqueryui.com/demos/effect/#default>.

Der `shake`-Effekt ist eine nette Sache, um mit Nachdruck anzuzeigen, dass ein Vorgang momentan nicht möglich ist. Wir können ihn beispielsweise einsetzen, wenn die `Resume`-Schaltfläche keine Wirkung hat:

```
$(document).ready(function() {
    $('<button>Resume</button>').click(function() {
        var $paused = $('ul:paused');
        if ($paused.length) {
            $paused.cycle('resume');
            $.cookie('cyclePaused', null);
        }
        else {
            $(this).effect('shake', {
                distance: 10,
                duration: 80
            });
        }
        return false;
    }).appendTo('#books-controls');
});
```

**Listing 7-11**

Unser neuer Code prüft die Länge von `'ul:paused'`, um zu entscheiden, ob es weitere pausierte Diashows gibt, die fortgesetzt werden können. Wenn ja, wird die *Resume*-Aktion von Cycle erneut aufgerufen. Wenn nicht, wird der shake-Effekt ausgeführt. Wir sehen hier, dass shake, wie die anderen Effekte auch, Optionen für Feineinstellungen besitzt. Hier setzen wir `distance` und `duration` des Effekts auf kleinere Werte als die Voreinstellung, damit die Schaltfläche bei einem Klick schnell hin- und herwackelt, so als wenn sie den Kopf schütteln und »Nein« sagen würde.

### 7.3.2 Interaktionskomponenten

Der nächste Teil unseres jQuery-UI-Puzzles sind die Interaktionskomponenten, die aus einem Satz kombinierbarer Verhaltensweisen bestehen und komplexe Interaktionen ermöglichen. Eine solche Komponente ist z.B. *Resizable*, womit der Benutzer die Größe jedes Elements durch einfache Ziehbewegungen verändern kann.

Einem Element eine Interaktion hinzuzufügen ist genauso einfach, wie eine gleichnamige Methode aufzurufen. Wenn wir z.B. die Buchtitel in der Größe verändern wollen, rufen wir `.resizable()` wie folgt auf:

```
$(document).ready(function() {  
    $('#books .title').resizable();  
});
```

**Listing 7-12**

Hierdurch fügen wir im Titelrahmen unten rechts einen Griff ein. Das Ziehen an diesem Griff verändert Höhe und Breite des Rahmens, wie im folgenden Screen-shot gezeigt:



Inzwischen erwarten wir schon, dass diese Methoden mit weiteren Optionen angepasst werden können. Wenn wir die Größenänderung auf die Vertikale beschränken möchten, erreichen wir das, indem wir festlegen, welche Art Griff hinzugefügt wird:

```
$(document).ready(function() {  
    $('#books .title').resizable({  
        handles: 's'  
    });  
});
```

*Listing 7-13*

Ist der Griff nur im Süden (unterhalb) der Region angebracht, kann nur die Höhe verändert werden, wie die folgende Abbildung zeigt:



#### Weitere interaktive Elemente

Die anderen interaktiven Elemente können ähnlich fein angepasst werden. Eine Liste einschließlich der Optionen finden Sie unter <http://jqueryui.com/demos/>.

### 7.3.3 Widgets

Zusätzlich zu diesen grundlegenden Interaktionskomponenten enthält jQuery UI eine Handvoll robuster Widgets für die Benutzerschnittstelle, die ohne großen Aufwand die gleichen Effekte erzielen, wie man sie aus ausgefeilten Desktop-Anwendungen kennt. Einige davon sind recht einfach: Das Widget *Button* verbessert z.B. Schaltflächen und Links auf einer Seite durch attraktive Formatierung und Rollover-Effekte.

Allen *Button*-Elementen einer Seite dieses Aussehen und Verhalten zu verschaffen, ist sehr einfach:

```
$(document).ready(function() {  
    $('button').button();  
});
```

*Listing 7-14*

Wenn das Stylesheet für das jQuery-UI-Theme *Smoothness* referenziert wird, erhalten die Schaltflächen eine glänzende, gewölbte Oberfläche wie in der folgenden Abbildung:

Pause      Resume

Wie andere UI-Widgets und Interaktionen hält *Button* verschiedene Optionen bereit. Vielleicht hätten wir gern passende Symbole für die beiden Schaltflächen. Das Widget *Button* bietet eine große Menge vordefinierter Symbole, die wir verwenden können. Dazu können wir unseren `.button()`-Aufruf in zwei Aufrufe aufteilen und für jeden ein Symbol zuweisen:

```
$(document).ready(function() {
    $('<button>Pause</button>').click(function() {
        // ...
    }).button({
        icons: {primary: 'ui-icon-pause'}
    }).appendTo('#books-controls');
    $('<button>Resume</button>').click(function() {
        // ...
    }).button({
        icons: {primary: 'ui-icon-play'}
    }).appendTo('#books-controls');
});
```

**Listing 7-15**

Die `primary`-Symbole, die wir angegeben haben, entsprechen Standard-Klassenbezeichnungen im jQuery-UI-Theme-Framework. Per Vorgabe werden `primary`-Symbole links der Schaltfläche angezeigt, während `secondary`-Symbole Rechts erscheinen:

|| Pause      ▶ Resume

Andere Widgets können wesentlich raffinierter sein. Das Widget *Slider* führt ein neues *Form*-Element ein, das einem HTML5-*Range*-Element gleicht, jedoch mit allen aktuellen Browsern kompatibel ist. Hierzu wird ein höheres Maß an Anpassung und Differenzierung benötigt:

```
$(document).ready(function() {
    $('<div id="slider"></div>').slider({
        min: 0,
        max: $('#books li').length - 1
    }).appendTo('#books-controls');
});
```

**Listing 7-16**

Ein Aufruf von `.slider()` transformiert ein einfaches `<div>`-Element in ein Slider-Widget. Das Widget wird durch Ziehen mit der Maus oder mit Cursortasten gesteuert:



In Listing 7–16 haben wir für den *Slider* einen Minimalwert von 0 angegeben und als Maximalwert den Index für das letzte Buch in unserer Diashow. Wir können auf diese Weise eine manuelle Steuerung der Diashow ermöglichen, indem wir Meldungen zwischen Diashow und *Slider* austauschen, wenn sich der Status ändert.

Um auf die Änderung eines *Slider*-Werts zu reagieren, können wir einen Handler an ein eigenes Ereignis binden, das durch *Slider* ausgelöst wird. Dieses Ereignis, *slide*, ist kein natives JavaScript-Ereignis, es verhält sich in unserem jQuery-Code aber genau wie eines. Solche Ereignisse zu überwachen ist jedoch so verbreitet, dass wir, statt *.bind()* explizit aufzurufen, unseren Ereignishandler einfach an den *.slider()*-Aufruf selbst anfügen:

```
$(document).ready(function() {
    $('<div id="slider"></div>').slider({
        min: 0,
        max: $('#books li').length - 1,
        slide: function(event, ui) {
            $('#books').cycle(ui.value);
        }
    }).appendTo('#books-controls');
});
```

**Listing 7–17**

Immer wenn der *slide*-Callback aufgerufen wird, wird der Parameter *ui* mit Informationen über das Widget gefüllt einschließlich des aktuellen Werts. Indem wir diesen Wert an das Cycle-Plug-in übergeben, können wir den dargestellten *Slider* verändern.

Um in umgekehrter Richtung zu kommunizieren, können wir den Cycle-Callback *before* verwenden, der vor jeder Diaüberblendung ausgelöst wird:

```
$(document).ready(function() {
    $('#books').cycle({
        timeout: 2000,
        speed: 200,
        pause: true,
        before: function() {
            $('#slider')
                .slider('value', $('#books li').index(this));
        }
    });
});
```

**Listing 7–18**

Innerhalb des before-Callbacks rufen wir die Methode `.slider()` erneut auf, dieses Mal mit 'value' als erstem Parameter, um den neuen Slider-Wert zu setzen. Im jQuery-UI-Jargon bezeichnen wir value als Methode von *Slider*, auch wenn er eigentlich durch die `.slider()`-Methode aufgerufen wird und nicht durch einen eigenen Methodennamen.

### Weitere Widgets

Jedes der jQuery-UI-Widgets besitzt mehrere Optionen, Ereignisse und Methoden. Eine vollständige Liste finden Sie unter <http://jqueryui.com/demos/>.

#### 7.3.4 JQuery-UI-ThemeRoller

Eine der spannendsten Funktionen der jQuery-UI-Bibliothek ist der ThemeRoller, eine webbasierte, interaktive Theme-Engine für UI-Widgets. Mit ThemeRoller sind hochgradig angepasste, professionell aussehende Elemente schnell und einfach erstellt. Die Schaltflächen und Silder, die wir gerade erstellt haben, gehören zum Standard-Theme. Dieses Theme wird vom ThemeRoller ausgegeben, wenn keine eigenen Einstellungen angegeben werden:



Um einen völlig anderen Satz von Formaten zu erstellen, müssen Sie nur <http://ui.jquery.com/themeroller/> besuchen, die verschiedenen Optionen wie gewünscht modifizieren und auf die Schaltfläche DOWNLOAD THEME klicken. Eine .zip-Datei mit Stylesheets und Bildern kann dann im Verzeichnis Ihrer Website entpackt werden. Wenn Sie beispielsweise andere Farben und Texturen auswählen, können Sie binnen Minuten ein neues, durchgängiges Aussehen für Ihre Schaltflächen, Symbole und Slider erhalten:



## **7.4 Zusammenfassung**

In diesem Kapitel haben Sie verschiedene Wege erkundet, wie Sie Plug-ins von Dritten in Ihre Webseiten integrieren können. Wir haben uns das Cycle-Plug-in, das Cookie-Plug-in und jQuery-UI genau angesehen und so die Vorgehensweisen kennengelernt, die bei Plug-ins wieder und wieder auftreten. Im nächsten Kapitel nutzen wir die jQuery-Plug-in-Architektur, um verschiedene eigene Plug-ins zu entwickeln.

## **7.5 Übungsaufgaben**

Um diese Übungsaufgaben durchzuarbeiten, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) heruntergeladen werden.

1. Verlängern Sie die Überblendungsdauer von Cycle um eine halbe Sekunde und ändern Sie die Animation so, dass jedes Dia vollständig ausgeblendet ist, bevor das nächste erscheint. Lesen Sie dazu in der Cycle-Dokumentation nach, welche Option verwendet werden muss.
2. Setzen Sie das Cookie für `cyclePaused` auf 30 Tage.
3. Beschränken Sie den Titelrahmen beim Vergrößern auf Schritte von 10 Pixeln.
4. Machen Sie die Bewegung des Sliders weicher, als es jetzt der Fall ist.
5. Halten Sie die Diashow an, wenn das letzte Dia gezeigt wurde, statt sie endlos laufen zu lassen. Deaktivieren Sie die Schaltflächen und Slider, wenn der Durchlauf vorbei ist.
6. Erstellen Sie ein neues UI-Theme für jQuery, das einen hellblauen Widget-Hintergrund mit dunkelblauem Text besitzt, und wenden Sie es auf unser Beispieldokument an.



## 8 Plug-ins entwickeln

Die zur Verfügung stehenden Drittanbieter-Plug-ins bieten eine Vielzahl von Optionen, um unseren Code zu verbessern, aber manchmal müssen wir noch einen Schritt weiter gehen. Wenn wir Code schreiben, der auch von anderen genutzt werden soll, müssen wir ihn als neues Plug-in verpacken. Erfreulicherweise ist das nicht viel aufwendiger, als den eigentlichen Code zu entwickeln.

In diesem Kapitel zeigen wir, wie verschiedene Arten von Plug-ins entwickelt werden: vom einfachen bis hin zum komplexen. Wir beginnen mit Plug-ins, die einfach nur neue globale Funktionen bereitstellen, und gehen dann zu unterschiedlichen jQuery-Objektmethoden über. Wir behandeln auch die Widget-Factory von jQuery UI, mit der Sie ohne viel Aufwand ausgefeilte Plug-ins entwickeln können.

### 8.1 Das Alias \$ innerhalb von Plug-ins verwenden

Wenn wir jQuery-Plug-ins schreiben, müssen wir natürlich annehmen, dass die jQuery-Bibliothek geladen ist. Wir können aber nicht davon ausgehen, dass das \$-Alias zur Verfügung steht. Rufen Sie sich in Erinnerung, dass die Methode `$.noConflict()` die Steuerung dieser Funktion abgeben kann. Um dem Rechnung zu tragen, sollten unsere Plug-ins jQuery-Methoden immer mit vollem Namen aufrufen oder \$ intern selbst definieren.

Besonders in längeren Plug-ins empfinden viele Entwickler, dass das Fehlen von \$ den Code schwerer lesbar macht. Um das zu verhindern, kann \$ lokal für den Gültigkeitsbereich des Plug-ins definiert werden, indem eine Funktion definiert und sofort aufgerufen wird. Die dafür erforderliche Syntax wird als *Immediately Invoked Function Expression (IIFE)* bezeichnet und sieht wie folgt aus:

```
(function($) {  
    // Hier steht der Code  
}) (jQuery);
```

Die umschließende Funktion nimmt einen einzelnen Parameter entgegen, an den wir das globale jQuery-Objekt übergeben. Der Parameter wird \$ genannt, sodass wir das \$-Alias ohne Konflikte verwenden können.

## 8.2 Neue globale Funktionen hinzufügen

Einige der integrierten Fähigkeiten von jQuery werden über den Mechanismus der *globalen Funktionen* bereitgestellt. Wie wir gesehen haben, sind dies *Methoden* innerhalb des jQuery-Objekts. In der Praxis stellen sie aber eher Funktionen in einem *jQuery-Namensraum* dar.

Ein besonderes Beispiel für diese Technik ist die Funktion `$.ajax()`. Alles, was `$.ajax()` macht, kann über eine normale globale Funktion erledigt werden, die einfach `ajax()` genannt wird. Diese Vorgehensweise kann aber zu Konflikten mit Funktionsnamen führen. Indem die Funktion in den jQuery-Namensraum eingebettet wird, müssen wir uns nur noch um Namenskonflikte mit anderen jQuery-Methoden kümmern.

Viele der globalen Funktionen in der jQuery-Bibliothek sind *Hilfsmethoden*, d.h., sie stellen Abkürzungen für häufig benötigte Aufgaben dar, die nicht leicht manuell durchzuführen sind. Gute Beispiele sind die Array-Funktionen `$.each()`, `$.map()` und `$.grep()`. Im Folgenden zeigen wir die Erstellung solcher Hilfsmethoden anhand von zwei trivialen Funktionen.

Um dem jQuery-Namensraum eine Funktion hinzuzufügen, weisen wir die neue Funktion einem jQuery-Objekt als *Eigenschaft* zu:

```
(function($) {
    $.sum = function(array) {
        // Hier folgt der Code
    };
})jQuery;
```

**Listing 8-1**

In jedem Code, der dieses Plug-in verwendet, können wir jetzt schreiben:

```
$.sum();
```

Dies funktioniert genau wie ein einfacher Funktionsaufruf und der Code innerhalb der Funktion wird ausgeführt.

Die Methode `sum()` nimmt ein Array entgegen, addiert die Werte darin und gibt das Ergebnis zurück. Der Code für unser Plug-in ist ziemlich kurz:

```
(function($) {
    $.sum = function(array) {
        var total = 0;
        $.each(array, function(index, value) {
            value = $.trim(value);
            value = parseFloat(value) || 0;
            total += value;
        });
        return total;
    };
})jQuery;
```

**Listing 8-2**

Beachten Sie, dass wir hier die Methode `$.each()` verwendet haben, um die einzelnen Werte des Arrays zu durchlaufen. Wir könnten auch eine `for()`-Schleife einsetzen, aber da wir sicher sein können, dass die jQuery-Bibliothek vor Aufruf unseres Plug-ins geladen wurde, können wir auch die gewohnte Syntax benutzen. Eine weitere nette Eigenschaft von `$.each()` ist, dass der erste Parameter auch ein Objekt sein darf.

Um unser Plug-in zu testen, erstellen wir eine einfache Tabelle mit Artikeln aus dem Supermarkt:

```
<table id="inventory">
  <thead>
    <tr class="one">
      <th>Product</th>
      <th>Quantity</th>
      <th>Price</th>
    </tr>
  </thead>
  <tfoot>
    <tr id="sum" class="two">
      <td>Total</td>
      <td></td>
      <td></td>
    </tr>
    <tr id="average">
      <td>Average</td>
      <td></td>
      <td></td>
    </tr>
  </tfoot>
  <tbody>
    <tr>
      <td>Spam</td>
      <td>4</td>
      <td>2.50</td>
    </tr>
    <tr>
      <td>Egg</td>
      <td>12</td>
      <td>4.32</td>
    </tr>
    <tr>
      <td>Gourmet Spam</td>
      <td>14</td>
      <td>7.89</td>
    </tr>
  </tbody>
</table>
```

Jetzt schreiben wir ein kurzes Skript, das die Zelle unten in der Tabelle mit der Summe aller Mengen füllt:

```
$(document).ready(function() {
    var $inventory = $('#inventory tbody');
    var quantities = $inventory.find('td:nth-child(2)')
        .map(function(index, qty) {
            return $(qty).text();
        })
        .get();
    var sum = $.sum(quantities);
    $('#sum').find('td:nth-child(2)').text(sum);
});
```

**Listing 8–3**

Ein Blick auf die fertige HTML-Seite zeigt, dass das Plug-in richtig funktioniert:

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
<i>Average</i>		

### 8.2.1 Mehrere Funktionen hinzufügen

Wenn unser Plug-in mehr als eine globale Funktion bereitstellen soll, könnten wir diese unabhängig voneinander deklarieren. Wir fügen ein Plug-in hinzu, das den Durchschnitt eines Arrays mit Zahlenwerten bildet:

```
(function($) {
    $.average = function(array) {
        if ($.isArray(array)) {
            return $.sum(array) / array.length;
        }
        return '';
    };
})(jQuery);
```

**Listing 8–4**

Aus Bequemlichkeit und Platzgründen verwenden wir das Plug-in `$.sum()`, das uns bei der Rückgabe der Werte für `$.average()` unterstützen soll. Um die Fehleranfälligkeit zu reduzieren, prüfen wir das Argument darauf, ob es ein Array ist, bevor wir mit der Berechnung beginnen.

Nachdem nun eine zweite Methode definiert ist, können wir sie wie gewohnt aufrufen:

```
$(document).ready(function() {
    var $inventory = $('#inventory tbody');
    var prices = $inventory.find('td:nth-child(3)')
        .map(function(index, qty) {
            return $(qty).text();
        }).get();

    var average = $.average(prices);
    $('#average').find('td:nth-child(3)')
        .text(average.toFixed(2));
});
```

***Listing 8–5***

Der Durchschnitt erscheint jetzt in der dritten Spalte, wie der folgenden Screenshot zeigt:

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
Total	30	
Average		4.90

Wir können auch eine alternative Syntax für die Definition unserer Funktionen verwenden, indem wir die Funktion `$.extend()` einsetzen:

```
(function($) {
    $.extend({
        sum: function(array) {
            var total = 0;
            $.each(array, function(index, value) {
                value = $.trim(value);
                value = parseFloat(value) || 0;
                total += value;
            });
            return total;
        },
        average: function(array) {
            if ($.isArray(array)) {
                return $.sum(array) / array.length;
            }
            return '';
        }
    });
})(jQuery);
```

***Listing 8–6***

Damit erhalten wir das gleiche Ergebnis.

Trotzdem könnten wir hier eine Verunreinigung des Namensraums produzieren. Auch wenn wir von den meisten JavaScript-Funktions- und Variablennamen durch Verwendung des jQuery-Namensraums abgeschirmt sind, können wir immer noch in Konflikt mit Funktionsnamen in anderen jQuery-Plug-ins geraten. Um das zu vermeiden, ist es am besten, alle globalen Funktionen eines Plug-ins folgendermaßen in einem einzelnen Objekt zu kapseln:

```
(function($) {
    $.mathUtils = {
        sum: function(array) {
            var total = 0;

            $.each(array, function(index, value) {
                value = $.trim(value);
                value = parseFloat(value) || 0;

                total += value;
            });
            return total;
        },
        average: function(array) {
            if ($.isArray(array)) {
                return $.mathUtils.sum(array) / array.length;
            }
            return '';
        }
    };
})(jQuery);
```

**Listing 8–7**

Dieses Vorgehen erzeugt im Wesentlichen einen weiteren Namensraum für unsere globalen Funktionen, `jQuery.mathUtils` genannt. Obwohl wir diese Funktionen informell immer noch als global bezeichnen können, sind es jetzt Methoden des Objekts `mathUtils`, das selbst wiederum eine Eigenschaft des globalen `jQuery`-Objekts ist. Daher müssen wir den Plug-in-Namen wie folgt in unseren Funktionsaufrufen verwenden:

```
$.mathUtils.sum(sum);
$.mathUtils.average(average);
```

Mit dieser Technik (und einem hinreichend eindeutigen Plug-in-Namen) sind wir vor Namensraumkollisionen in unseren globalen Funktionen geschützt. Jetzt sind wir bestens mit den Grundlagen der Plug-in-Entwicklung vertraut. Nachdem wir unsere Funktionen in einer Datei namens `jquery.mathutils.js` gesichert haben, können wir das Skript einbinden und die Funktionen aus anderen Skripten auf der Seite verwenden.

#### Einen Namensraum wählen

Funktionen, die nur für den persönlichen Gebrauch gedacht sind, gehören eher in den globalen Namensraum des eigenen Projekts. Statt jQuery zu verwenden, können wir auch ein eigenes globales Objekt einsetzen. Wir können z.B. ein globales Objekt namens `ljq` verwenden und die Methoden `ljq.mathUtils.sum()` und `ljqmath-Utils.average()` definieren statt `$.mathUtils.sum()` und `$.mathUtils.average()`. Auf diese Weise verhindern wir jede mögliche Namensraumkollision mit Plug-ins von Drittanbietern, die wir einsetzen könnten.

Jetzt haben wir den Schutz von Namensräumen und die sichere Verfügbarkeit von Bibliotheken kennengelernt, die jQuery-Plug-ins bieten. Es handelt sich dabei aber nur um organisatorische Vorteile. Um die Leistungsfähigkeit der jQuery-Plug-ins wirklich zu nutzen, müssen wir lernen, wie wir in einzelnen jQuery-Objektinstanzen neue *Methoden* erstellen.

### 8.3 JQuery Objektmethoden hinzufügen

Die meiste in jQuery vorhandene Funktionalität beruht auf Objektinstanzmethoden und hier können Plug-ins ebenfalls glänzen. Immer wenn wir eine Funktion schreiben, die als Teil des DOM fungiert, ist es häufig sinnvoll, diese als Objektmethode zu erstellen.

Wir haben gesehen, dass das Hinzufügen globaler Funktionen erfordert, das jQuery-Objekt um neue Methoden zu erweitern. Eine Instanzmethode hinzuzufügen funktioniert genauso, dabei erweitern wir aber das Objekt `jQuery.fn`:

```
jQuery.fn.myMethod = function() {  
    alert('Nothing happens.');//  
};
```

Das Objekt `jQuery.fn` ist ein Alias für `jQuery.prototype` und fungiert als Abkürzung.

Wir können diese neue Methode dann aus unserem Code über jeden Selektorausdruck aufrufen.

```
$('.div').myMethod();
```

Unser Alarm wird angezeigt (einmal je `<div>` im Dokument), wenn wir die Methode aufrufen. Wir hätten auch eine globale Funktion schreiben können, da wir keine DOM-Knoten eingesetzt haben. Die sinnvolle Implementierung einer Methode basiert auf ihrem *Kontext*.

### 8.3.1 Kontext von Objektmethoden

In jeder Plug-in-Methode wird das Schlüsselwort `this` auf das aktuelle jQuery-Objekt gesetzt. Daher können wir mit `this` alle integrierten jQuery-Methoden aufrufen oder die DOM-Knoten extrahieren und sie bearbeiten. Um herauszufinden, was wir mit dem Objektkontext anstellen können, schreiben wir ein kleines Plug-in, mit dem wir die Klassen auf den dazugehörigen Elementen manipulieren können.

Unsere neue Methode übernimmt zwei Klassennamen und tauscht mit jedem Aufruf, welche Klasse mit welchem Element verknüpft ist. Da jQuery UI eine robuste Methode `.switchClass()` besitzt, die sogar eine Animation der Klassenänderung unterbindet, stellen wir eine einfache Implementierung zu Demonstrationszwecken zur Verfügung:

```
// Unvollständiger Code
(function($) {
    $.fn.swapClass = function(class1, class2) {
        if (this.hasClass(class1)) {
            this.removeClass(class1).addClass(class2);
        }
        else if (this.hasClass(class2)) {
            this.removeClass(class2).addClass(class1);
        }
    };
})(jQuery);

$(document).ready(function() {
    $('table').click(function() {
        $('tr').swapClass('one', 'two');
    });
});
```

**Listing 8-8**

In unserem Plug-in testen wir zuerst, ob `class1` im passenden Element vorhanden ist, und ersetzen diese durch `class2`, wenn wir fündig werden. Ist keine der beiden Klassen vorhanden, passiert gar nichts.

Im Code, der unser Plug-in verwendet, binden wir einen `click`-Handler an die Tabelle, der für jede Zeile `.swapClass()` aufruft, wenn die Tabelle angeklickt wird. Wir erwarten, dass sich dadurch die Klasse in der Header-Zeile von `one` auf `two` ändert und die Klasse der Summenzeile von `two` auf `one`. Das geschieht jedoch nicht:

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average		4.90

Jede Zeile hat Klasse `two` erhalten. Um dies in Ordnung zu bringen, müssen wir jQuery-Objekte mit mehreren ausgewählten Elementen korrekt behandeln.

### 8.3.2 Implizite Iteration

Wir müssen daran denken, dass ein jQuery-Selektorausdruck immer null, eines oder mehrere Elemente betreffen kann. All diese Szenarien müssen berücksichtigt werden, wenn wir eine Plug-in-Methode erstellen. In unserem Beispiel rufen wir `.hasClass()` auf, das nur das erste passende Element untersucht. Stattdessen müssen wir jedes Element unabhängig prüfen und entsprechend reagieren.

Der einfachste Weg, für ein ordnungsgemäßes Verhalten bei verschiedenen vielen passenden Elementen zu sorgen, besteht darin, `.each()` immer im Methodenkontext aufzurufen. Dadurch erreichen wir ein Verhalten, das implizite Iteration genannt wird und das wichtig für die Konsistenz zwischen Plug-ins und integrierten Methoden ist.

Innerhalb des `.each()`-Aufrufs wird auch das DOM-Element referenziert, sodass wir unseren Code anpassen können, um auf Klassen zu prüfen und sie jedem einzelnen Element zuzuweisen, wie im folgenden Codefragment gezeigt wird:

```
(function($) {
    $.fn.swapClass = function(class1, class2) {
        this.each(function() {
            var $element = $(this);
            if ($element.hasClass(class1)) {
                $element.removeClass(class1).addClass(class2);
            }
            else if ($element.hasClass(class2)) {
                $element.removeClass(class2).addClass(class1);
            }
        });
    };
})(jQuery);
```

**Listing 8-9**

### Die Bedeutung von this

Vorsicht! Das Schlüsselwort `this` bezieht sich auf ein jQuery-Objekt innerhalb des Rumpfs der Objektmethode, aber innerhalb des `.each()`-Aufrufs auf ein DOM-Element.

Wenn wir jetzt auf die Tabelle klicken, werden die Klassen vertauscht, ohne dass die Zeilen beeinträchtigt werden, denen keine Klasse zugewiesen ist:

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average		4.90

### 8.3.3 Verkettete Methoden

Zusätzlich zur impliziten Iteration sollten sich jQuery-Benutzer auf das Verhalten verketteter Methoden verlassen können. Das bedeutet, dass wir aus allen Plug-in-Methoden ein jQuery-Objekt zurückgeben müssen, es sei denn, die Methode dient klar dazu, eine andere Information bereitzustellen. Das zurückgegebene jQuery-Objekt ist normalerweise einfach das in `this` bereitgestellte.

Wenn wir `.each()` verwenden, um über `this` zu iterieren, können wir das Ergebnis einfach zurückgeben, wie im folgenden Beispiel gezeigt:

```
(function($) {
    $.fn.swapClass = function(class1, class2) {
        return this.each(function() {
            var $element = $(this);
            if ($element.hasClass(class1)) {
                $element.removeClass(class1).addClass(class2);
            }
            else if ($element.hasClass(class2)) {
                $element.removeClass(class2).addClass(class1);
            }
        });
    });
})(jQuery);
```

**Listing 8-10**

Wenn wir früher `.swapClass()` aufgerufen haben, mussten wir eine neue Anweisung beginnen, um mit den Elementen etwas anderes zu machen. Durch die `return`-Anweisung können wir unsere Plug-in-Methode beliebig mit integrierten Methoden verketten.

## 8.4 Methodenparameter

In Kapitel 7, »*Plug-ins verwenden*«, haben wir gesehen, dass einige Plug-ins mit Parametern sehr präzise konfiguriert werden können und dass ein intelligent programmiertes Plug-in uns dabei hilft, sinnvolle Voreinstellungen bereitzustellen, die überschrieben werden können. Wenn wir eigene Plug-ins programmieren, sollten wir das ebenfalls ermöglichen.

Als Beispiel beginnen wir mit einer Plug-in-Methode, die einen Schatten über ein Element legt. Das kann mit verschiedenen fortgeschrittenen CSS-Techniken erreicht werden, hier jedoch werden wir einen größeren Ansatz verwenden: Wir werden eine Anzahl von Elementen generieren, die teiltransparent sind und sich in verschiedenen Positionen auf der Seite überlagern:

```
(function($) {
    $.fn.shadow = function() {
        return this.each(function() {
            var $originalElement = $(this);
            for (var i = 0; i < 5; i++) {
                $originalElement
                    .clone()
                    .css({
                        position: 'absolute',
                        left: $originalElement.offset().left + i,
                        top: $originalElement.offset().top + i,
                        margin: 0,
                        zIndex: -1,
                        opacity: 0.1
                    })
                    .appendTo('body');
            }
        });
    });
})(jQuery);
```

**Listing 8-11**

Für jedes mit dieser Methode aufgerufene Element erzeugen wir eine Reihe von Kopien mit unterschiedlicher Deckkraft. Diese Kopien werden mit variablen Offsets absolut zum Ursprungselement positioniert. Für den Moment übernimmt unser Plug-in keine Parameter, sodass der Aufruf einfach ist:

```
$(document).ready(function() {
    $('h1').shadow();
});
```

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average		4.90

Dann geben wir der Plug-in-Methode etwas mehr Flexibilität mit. Die Funktion der Methode basiert auf mehreren numerischen Werten, die der Benutzer möglicherweise beeinflussen möchte. Wir können daraus *Parameter* machen, sodass sie bei Bedarf geändert werden können.

#### 8.4.1 Parameter-Maps

Wir haben in vielen Beispielen in der jQuery-API Maps gesehen, die als Methodenparameter verwendet werden. Dies kann für den Benutzer eines Plug-ins viel angenehmer sein als unsere einfache Parameterliste. Eine *Map* bietet für jeden Parameter eine Beschreibung an, außerdem ist die Reihenfolge der Werte unwichtig. Zusätzlich sollten wir die jQuery-API in unseren Plug-ins nachbilden, wann immer wir können. Das erhöht die Konsistenz und damit den Programmierkomfort.

```
(function($) {
    $.fn.shadow = function(options) {
        return this.each(function() {
            var $originalElement = $(this);
            for (var i = 0; i < options.copies; i++) {
                $originalElement
                    .clone()
                    .css({
                        position: 'absolute',
                        left: $originalElement.offset().left + i,
                        top: $originalElement.offset().top + i,
                        margin: 0,
                    })
                    .appendTo(this);
            }
        });
    };
});
```

```
        zIndex: -1,
        opacity: options.opacity
    })
    .appendTo('body');
}
});
});
}) (jQuery);
```

**Listing 8-12**

Die Anzahl der Kopien und deren Deckkraft können jetzt angepasst werden. In unserem Plug-in wird jeder Wert als Eigenschaft des `options`-Arguments der Funktion ausgelesen.

Wenn wir die Methode aufrufen, müssen wir die Parameter jetzt mit angeben:

```
$(document).ready(function() {
    $('h1').shadow({
        copies: 3,
        opacity: 0.25
    });
});
```

Die Konfigurationsmöglichkeiten sind eine Verbesserung, aber jetzt müssen wir jedes Mal beide Werte mit angeben. Dieses Problem beheben wir im folgenden Abschnitt.

#### 8.4.2 Voreinstellungen für Parameterwerte

Je mehr Parameter wir für eine Methode angeben können, desto geringer wird die Wahrscheinlichkeit, dass wir immer auch jeden Wert setzen wollen. Eine sinnvolle Vorbelegung von Werten kann die Schnittstelle zu einem Plug-in wesentlich angenehmer gestalten. Zum Glück hilft uns die Verwendung einer Map für unsere Parameter dabei weiter. Es ist einfach, Elemente der Map wegzulassen und durch eine Voreinstellung zu ersetzen.

```
(function($) {
    $.fn.shadow = function(opts) {
        var defaults = {
            copies: 5,
            opacity: 0.1
        };
        var options = $.extend(defaults, opts);
        // ...
    };
}) (jQuery);
```

**Listing 8-13**

Hier haben wir innerhalb unserer Methodendefinition eine neue Map namens `defaults` definiert. Die Hilfsfunktion `$.extend()` ermöglicht es uns, die als Argument übergebene Map `opts` zu nehmen und damit die Elemente in `defaults` zu überschreiben. So bleiben nicht überschriebene Voreinstellungen erhalten.

Noch immer rufen wir unsere Methode mit einer Map auf, aber jetzt brauchen wir nur die Parameter anzugeben, die von der Voreinstellung abweichen sollen, so wie hier:

```
$(document).ready(function() {
    $('h1').shadow({
        copies: 3
    });
});
```

Nicht angegebene Parameter erhalten den Vorgabewert. Die Methode `$.extend()` akzeptiert sogar leere Werte. Wenn die Voreinstellungen also alle geeignet sind, kann die Methode einfach und ohne Fehler aufgerufen werden:

```
$(document).ready(function() {
    $('h1').shadow();
});
```

#### 8.4.3 Callback-Funktionen

Natürlich können die Parameter einer Methode komplizierter sein als einfache Zahlenwerte. Ein in der jQuery-API häufig anzutreffender Parametertyp ist die *Callback-Funktion*. Callback-Funktionen können einem Plug-in eine große Flexibilität verleihen, ohne dass bei ihrer Programmierung viele Vorbereitungsarbeiten erforderlich wären.

Um eine Callback-Funktion in unserer Methode zu verwenden, müssen wir das Funktionsoobjekt einfach als Parameter akzeptieren und die Funktion, wenn nötig, in unserer Implementierung der Methode aufrufen. Zum Beispiel können wir unsere `shadow()`-Methode erweitern, sodass der Benutzer die Position des Schattens relativ zum Text anpassen kann:

```
(function($) {
    $.fn.shadow = function(opts) {
        var defaults = {
            copies: 5,
            opacity: 0.1,
            copyOffset: function(index) {
                return {x: index, y: index};
            }
        };
        var options = $.extend(defaults, opts);
```

```
return this.each(function() {
    var $originalElement = $(this);
    for (var i = 0; i < options.copies; i++) {
        var offset = options.copyOffset(i);
        $originalElement
            .clone()
            .css({
                position: 'absolute',
                left: $originalElement.offset().left + offset.x,
                top: $originalElement.offset().top + offset.y,
                margin: 0,
                zIndex: -1,
                opacity: options.opacity
            })
            .appendTo('body');
    }
});
})(jQuery);
```

**Listing 8-14**

Jede Kopie des Schattens hat einen anderen Offset zum Originaltext. Vorher bestand dieser Offset einfach aus dem Index der Kopie. Jetzt können wir den Offset mittels der Funktion `copyOffset()` berechnen, die eine Option darstellt, die der Benutzer überschreiben kann. So können wir beispielsweise für den Offset negative Werte für beide Dimensionen verwenden:

```
$(document).ready(function() {
    $('h1').shadow({
        copyOffset: function(index) {
            return {x: -index, y: -2 * index};
        }
    });
});
```

Dadurch wird der Schatten nach links oben gerichtet statt nach rechts unten:

Inventory		
Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average		4.90

Der Callback ermöglicht einfache Änderungen der Schattenrichtung und eine ausgelügelte Positionierung, wenn der Benutzer des Plug-ins den passenden Callback verwendet. Wenn der Callback nicht angegeben wird, tritt wieder das vorgegebene Verhalten auf.

#### 8.4.4 Anpassbare Voreinstellungen

Wir können die Verwendung unserer Plug-ins noch angenehmer gestalten, indem wir wie gezeigt sinnvolle Voreinstellungen für unsere Methodenparameter wählen. Manchmal ist es jedoch schwierig, vorherzusehen, was ein sinnvoller Vorgabewert ist. Wenn ein Skript unser Plug-in mehrfach mit einem abweichenden Parametersatz als unserem aufruft, kann die Option, diese Voreinstellungen anzupassen, eine Menge Programmieraufwand einsparen.

Um die Voreinstellungen anpassbar zu gestalten, müssen wir sie aus unserer Methodendefinition entfernen und an einem Ort ablegen, der vom externen Code erreichbar ist:

```
(function($) {
    $.fn.shadow = function(opts) {
        var options = $.extend({}, $.fn.shadow.defaults, opts);
        // ...
    };

    $.fn.shadow.defaults = {
        copies: 5,
        opacity: 0.1,
        copyOffset: function(index) {
            return {x: index, y: index};
        }
    };
})(jQuery);
```

**Listing 8-15**

Die Voreinstellungen befinden sich nun im Namensraum des Plug-ins und können direkt mit der Funktion `$.fn.shadow.defaults` angesprochen werden. Unser Aufruf von `$.extend()` musste sich entsprechend ändern. Da wir jetzt dieselbe Voreinstellungs-Map für jeden Aufruf von `.shadow()` verwenden, darf `$.extend()` diese nicht mehr verändern. Stattdessen bieten wir als erstes Argument für `$.extend()` eine leere Map {}, und dieses neue Objekt kann jetzt verändert werden.

Jetzt kann Code, der unser Plug-in einsetzt, die Voreinstellungen ändern, die für alle folgenden Aufrufe von `.shadow()` verwendet werden. Die Optionen können außerdem auch dann noch angegeben werden, wenn die Methode aufgerufen wird:

```
$(document).ready(function() {  
    $.fn.shadow.defaults.copies = 10;  
    $('h1').shadow({  
        copyOffset: function(index) {  
            return {x: -index, y: index};  
        }  
    });  
});
```

Dieses Skript erzeugt einen Schatten aus 10 Kopien des Elements, da das der neue Vorgabewert ist, es positioniert den Schatten aber auch nach links unten entsprechend dem `copyOffset`-Callback, der mit dem Methodenaufruf erfolgt:

Product	Quantity	Price
Spam	4	2.50
Egg	12	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average	4.90	

## 8.5 Die Widget-Factory von jQuery UI

Wie wir in Kapitel 7 gesehen haben, bietet jQuery UI eine Auswahl an Widgets – Plug-ins, die als bestimmtes UI-Element vorhanden sind, wie z.B. Schaltflächen oder Slider. Diese Widgets stellen eine sehr konsistente API für JavaScript-Programmierer zur Verfügung, wodurch das Erlernen viel leichter wird. Wenn ein von uns geschriebenes Plug-in ein neues Element für die Benutzerschnittstelle erzeugt, ist es häufig eine gute Entscheidung, die jQuery-UI-Bibliothek um ein Widget-Plug-in zu ergänzen.

Ein Widget ist eine aufwendige Funktion, glücklicherweise sind wir aber bei der Erstellung nicht vollständig auf uns selbst gestellt. Der jQuery-UI-Kern enthält eine Factory-Methode namens `$.widget()`, die uns einen großen Teil der Arbeit abnimmt. Verwenden wir die Factory, können wir sicher sein, dass unser Code dem API-Standard entspricht, den alle jQuery-UI-Widgets verwenden.

Plug-ins, die wir mit der Widget-Factory erstellen, haben viele nette Funktionen. All diese Vorteile (und noch mehr!) erhalten wir ohne viel eigenen Aufwand:

1. Das Plug-in hat einen Status. Das bedeutet, dass wir es untersuchen, verändern oder seine Auswirkungen komplett rückgängig machen können, auch wenn es schon angewendet wurde.
2. Benutzeroptionen werden mit anpassbaren Voreinstellungen automatisch zusammengeführt.

3. Mehrere Plug-in-Methoden werden nahtlos in einer einzelnen jQuery-Methode zusammengefasst. Untermethoden werden mittels Zeichenketten gekennzeichnet.
4. Eigene vom Plug-in ausgelöste Ereignishandler erhalten Zugriff auf die Daten der Widget-Instanz.

Diese Vorteile sind so angenehm, dass wir uns wünschen werden, die Widget-Factory für alle komplexen Plug-ins einsetzen zu können, seien sie UI-bezogen oder nicht.

### 8.5.1 Ein Widget erstellen

In unserem Beispiel erstellen wir ein Plug-in, das Elementen eigene Quickinfos hinzufügt. Eine einfache Implementierung erzeugt für jedes Element auf der Seite, das eine Quickinfo erhalten soll, einen <div>-Container und positioniert ihn neben dem Element, wenn der Mauszeiger über das Ziel fährt. Wir sehen uns den folgenden lauffähigen Plug-in-Code an und gehen ihn dann schrittweise durch.

```
(function($) {
    $.widget('ljq.tooltip', {
        _create: function() {
            this._tooltipDiv = $('<div></div>')
                .addClass('ljq-tooltip-text ' +
                    'ui-widget ui-state-highlight ui-corner-all')
                .hide().appendTo('body');

            this.element
                .addClass('ljq-tooltip-trigger')
                .bind('mouseenter.ljq-tooltip',
                    $.proxy(this._open, this))
                .bind('mouseleave.ljq-tooltip',
                    $.proxy(this._close, this));
        },
        _open: function() {
            var elementOffset = this.element.offset();
            this._tooltipDiv.css({
                left: elementOffset.left,
                top: elementOffset.top + this.element.height()
            }).text(this.element.data('tooltip-text'));
            this._tooltipDiv.show();
        },
        _close: function() {
            this._tooltipDiv.hide();
        }
    });
})(jQuery);
```

**Listing 8-16**

Ein Plug-in wird von der Widget-Factory immer dann erzeugt, wenn `$.widget()` aufgerufen wird. Diese Funktion übernimmt den Namen und eine Map mit den Widget-Eigenschaften. Der Name des Widgets muss einen Namensraum enthalten. Hier haben wir den Namensraum `1jq` und den Plug-in-Namen `tooltip` gewählt. Deshalb wird unser Plug-in durch `.tooltip()` von einem jQuery-Objekt aufgerufen.

In unserem Beispiel sind alle Widget-Eigenschaften Funktionen. Die Funktionsnamen beginnen mit einem Unterstrich, weil sie *privat* sind. Wir werden öffentliche Funktionen später besprechen. Die erste Funktion, `_create`, ist etwas speziell und wird durch die Widget-Factory erstellt. `.tooltip()` wird einmal je passendem Element im jQuery-Objekt aufgerufen.

Innerhalb dieser Erstellungsfunktion müssen wir unsere Quickinfo für die spätere Anzeige einrichten. Dazu erstellen wir ein neues `<div>`-Element und fügen es dem Dokument hinzu. Wir speichern das erstellte Element in `this._tooltipDiv` zur späteren Verwendung.

Im Kontext unserer Funktion bezieht sich `this` auf die aktuelle Instanz des Widgets und wir können diesem Objekt alle gewünschten Eigenschaften hinzufügen. Das Objekt besitzt einige eingebaute Eigenschaften, die für uns ebenfalls nützlich sein können. Insbesondere über gibt uns `this.element` ein jQuery-Objekt, das auf das ursprünglich ausgewählte Element zeigt.

Wir verwenden `this.element`, um die Handler `mouseenter` und `mouseleave` an das Trigger-Element der Quickinfo zu binden. Diese Handler sollen die Quickinfo öffnen, wenn die Maus über den Auslöser fährt, und sie schließen, wenn die Maus ihn verlässt. Die Ereignisse sind im selben Namensraum wie unser Plug-in. Wie in Kapitel 3, »*Ereignisbehandlung*«, besprochen, ist es durch den Namensraum einfacher, Ereignishandler einzuführen und zu entfernen, ohne anderem Code auf die Füße zu treten, der ebenfalls Handler an diese Elemente binden will.

Die `.bind()`-Aufrufe enthalten eine weitere Funktion, die neu für uns ist: Die Ereignishandler werden durch die Funktion `$.proxy()` geschleust. Diese Funktion ändert den Bezug von `this` in einer Methode, sodass wir die Widget-Instanz leicht in der Funktion `_open` referenzieren können.

Die Funktionen `_open` und `_close` sind selbsterklärend. Es handelt sich dabei nicht um besondere Namen, sondern sie bezeichnen eher, dass wir beliebige private Funktionen in unserem Widget erzeugen können, solange ihre Namen mit einem Unterstrich beginnen. Wenn die Quickinfo geöffnet wird, positionieren wir sie mit CSS und zeigen sie an. Wird sie geschlossen, verbergen wir sie einfach.

Während des Öffnens müssen wir Informationen in die Quickinfo schreiben. Dafür verwenden wir die Methode `.data()`, die Zusatzdaten für ein Element lesen und schreiben kann, hier jedoch die Werte aus dem Attribut `data-tooltip-text` zurückgibt.

Mit unserem Plug-in an der richtigen Stelle erzeugt der Code `$('a').tooltip()` eine Quickinfo, die angezeigt wird, wenn die Maus sich über einem Anker befindet.

The screenshot shows a table titled "Inventory". The table has columns for Product, Quantity, and Price. There are three rows of data: "Spam" (Quantity 4, Price 2.50), "Nutritious and delicious!" (Quantity 14, Price 4.32), and "Gourmet Spam" (Quantity 14, Price 7.89). Below the data is a summary row with "Total" (Quantity 30, Price 4.90) and an "Average" row (Price 4.90). A tooltip is displayed over the "Nutritious and delicious!" row, containing the text "Nutritious and delicious!".

Product	Quantity	Price
Spam	4	2.50
Nutritious and delicious!	14	4.32
Gourmet Spam	14	7.89
<b>Total</b>	<b>30</b>	
Average		4.90

Das Plug-in ist nicht sehr lang, aber dicht gepackt mit ausgefeilten Konzepten. Damit sich diese lohnen, müssen wir unser Widget jetzt statusbehaftet bekommen.

### 8.5.2 Widgets entfernen

Wir haben gesehen, dass die Widget-Factory eine neue jQuery-Methode erzeugt, in unserem Fall `.tooltip()`, die ohne Argumente aufgerufen werden kann, um das Widget einer Reihe von Elementen zuzuweisen. Doch diese Methode kann noch viel mehr tun. Wenn wir der Methode ein Zeichenkettenargument übergeben, benennt es die *Untermethode* mit dem entsprechenden Namen.

Eine der eingebauten Untermethoden wird `destroy` genannt. Ein Aufruf von `.tooltip('destroy')` entfernt das Quickinfo-Widget von der Seite. Die Widget-Factory erledigt die meiste Arbeit, wenn wir jedoch Teile des Dokuments innerhalb von `_create` verändert haben (wie hier, weil wir den Quickinfo-Text `<div>` erstellt haben), müssen wir manuell aufräumen.

```
destroy: function() {
    this._tooltipDiv.remove();
    this.element
        .removeClass('ljq-tooltip-trigger')
        .unbind('.ljq-tooltip');
    $.Widget.prototype.destroy.apply(this, arguments);
},
```

**Listing 8-17**

Dieser neue Code wird dem Widget als neue Eigenschaft hinzugefügt. Beachten Sie, dass `destroy` nicht mit einem Unterstrich beginnt. Es handelt sich um eine öffentliche Untermethode, die wir mit `.tooltip('destroy')` aufrufen können. Die Funktion macht dann die vorgenommenen Änderungen rückgängig und ruft den Prototyp von `destroy` auf, sodass automatisch aufgeräumt wird.

### 8.5.3 Widgets aktivieren und deaktivieren

Zusätzlich zur dauerhaften Entfernung kann ein Widget auch zeitweise deaktiviert und später wieder aktiviert werden. Die integrierten Untermethoden `enable` und `disable` helfen uns dabei, den Wert von `this.options.disabled` auf `true` oder `false` zu setzen. Alles, was wir tun müssen, um diese Untermethoden zu unterstützen, ist, folgenden Wert zu prüfen, bevor unser Widget eine Aktion vornimmt:

```
_open: function() {
    if (!this.options.disabled) {
        var elementOffset = this.element.offset();
        this._tooltipDiv.css({
            left: elementOffset.left,
            top: elementOffset.top + this.element.height()
        }).text(this.element.data('tooltip-text'));
        this._tooltipDiv.show();
    }
},
```

*Listing 8-18*

Mit dieser Zusatzprüfung verschwinden die Quickinfos, sobald `.tooltip('disable')` aufgerufen wird, und erscheinen nach `.tooltip('enable')` wieder.

### 8.5.4 Widget-Optionen übernehmen

Jetzt ist es an der Zeit, unser Widget anpassbar zu machen. Wie wir bei der Erstellung des Plug-ins `.shadow()` gesehen haben, ist es eine gute Angewohnheit, anpassbare Vorgabewerte für ein Widget zu liefern und diese Voreinstellungen dann mit Werten des Benutzers zu überschreiben. Fast alle diese Aufgaben werden durch die Widget-Factory übernommen. Wir müssen nur eine `options`-Eigenschaft liefern:

```
options: {
    offsetX: 10,
    offsetY: 10,
    content: function() {
        return $(this).data('tooltip-text');
    }
},
_open: function() {
    if (!this.options.disabled) {
        var elementOffset = this.element.offset();
        this._tooltipDiv.css({
            left: elementOffset.left + this.options.offsetX,
            top: elementOffset.top + this.element.height()
            + this.options.offsetY
        })
    }
},
```

```
        }).text(this.options.content.call(this.element[0]));
        this._tooltipDiv.show();
    }
},
```

**Listing 8-19**

Die Eigenschaft `options` ist eine einfache Map. Alle gültigen Optionen für unser Widget sollten enthalten sein, sodass der Benutzer keine eigenen angeben muss. Hier liefern wir für die Quickinfo X- und Y-Koordinaten relativ zum Auslöser sowie eine Funktion, die den Quickinfo-Text für jedes Element generiert.

Der einzige Teil unseres Codes, der diese Optionen prüfen muss, ist `_open`, die anderen Funktionen bleiben also unverändert. Innerhalb einer Untermethode wie `_open` können wir auf die Eigenschaften mittels `this.options` zugreifen. So erhalten wir immer den richtigen Wert für die Option: die Voreinstellung oder den überschriebenen Wert, wenn der Benutzer einen eigenen angegeben hat.

Wir können unser Widget immer noch ohne Argumente als `.tooltip()` hinzufügen und erhalten immer noch das Standardverhalten. Wir können aber auch Optionen angeben, die das Standardverhalten überschreiben: `.tooltip({offsetX: -10, offsetY: 25})`. Die Widget-Factory lässt uns diese Optionen sogar verändern, nachdem das Widget instanziiert wurde: `.tooltip('option', 'offsetX', 20)`. Wenn das nächste Mal auf die Option zugegriffen wird, erscheint der neue Wert.

**Auf veränderte Optionen reagieren**

Wenn wir sofort auf eine veränderte Option reagieren müssen, können wir unserem Widget eine `_setOption`-Funktion hinzufügen, die die Änderung bearbeitet und danach die Standardimplementierung von `_setOption` aufruft.

### 8.5.5 Untermethoden hinzufügen

Die eingebauten Untermethoden sind bequem, oft aber möchten wir dem Benutzer unseres Plug-ins aber mehr Verknüpfungspunkte anbieten. Wir haben bereits gesehen, wie neue private Funktionen innerhalb unseres Widgets erstellt werden. Öffentliche Funktionen (Untermethoden) zu erstellen funktioniert genauso, nur dass die Namen der Widget-Eigenschaften nicht mit einem Unterstrich beginnen dürfen. Wir können damit Untermethoden erzeugen, die die Quickinfo einfach manuell öffnen und schließen:

```
open: function() {
    this._open();
},
close: function() {
    this._close();
},
```

**Listing 8-20**

Das war's! Durch das Hinzufügen öffentlicher Untermethoden, die die privaten Funktionen aufrufen, können wir die Quickinfo jetzt mit `.tooltip('open')` öffnen und mit `.tooltip('close')` schließen. Die Widget-Factory übernimmt all die kleinen Aufgaben für uns, wie z.B. das Sicherstellen, dass die Verkettung weiter funktioniert, selbst wenn wir nichts von der Untermethode zurückgeben.

### 8.5.6 Widget-Ereignisse auslösen

Ein gutes Plug-in erweitert nicht nur jQuery, sondern bietet auch anderem Code die Möglichkeit, das Plug-in selbst zu erweitern. Ein einfacher Weg für diese Erweiterbarkeit ist es, zum Plug-in gehörende Ereignisse zu unterstützen. Mithilfe der Widget-Factory wird daraus ein einfacher Prozess:

```
_open: function() {
    if (!this.options.disabled) {
        var elementOffset = this.element.offset();
        this._tooltipDiv.css({
            left: elementOffset.left + this.options.offsetX,
            top: elementOffset.top + this.element.height()
                + this.options.offsetY
        }).text(this.options.content.call(this.element[0]));
        this._tooltipDiv.show();
        this._trigger('open');
    }
},
_close: function() {
    this._tooltipDiv.hide();
    this._trigger('close');
}
```

*Listing 8-21*

Der Aufruf von `this._trigger()` in einer unserer Funktionen ermöglicht es Code, auf das neue benutzerdefinierte Ereignis zu warten. Der Ereignisname erhält als Präfix den Namen des Widgets, sodass wir uns nicht um Konflikte mit anderen Ereignissen sorgen müssen. Wenn wir in unserer Funktion `this._trigger('open')` zum Öffnen der Quickinfo aufrufen, wird z.B. jedes Mal das Ereignis namens `tooltipopen` ausgelöst. Wir können auf dieses Ereignis lauschen, indem wir im Element `.bind('tooltipopen')` aufrufen.

Wir befinden uns hier immer noch an der Oberfläche, was die Möglichkeiten eines ausgefeilten Widget-Plug-ins betreffen. Allerdings verfügen wir jetzt über die Werkzeuge, um ein Widget zu erstellen, das Funktionen hat und dem Standard genügt, den Benutzer der jQuery UI erwarten.

## 8.6 Designempfehlungen für Plug-ins

Nachdem wir beim Erstellen von Plug-ins die üblichen Verfahren zur Erweiterung von jQuery und jQuery UI behandelt haben, können wir das Gelernte durch eine Liste mit Empfehlungen vertiefen und ergänzen.

- Schützen Sie das \$-Alias vor Einflüssen durch andere Bibliotheken, indem Sie jQuery verwenden oder \$ in eine *Immediately Invoked Function Expression* (IIFE) übertragen, sodass es als lokale Variable genutzt werden kann.
- Egal, ob Sie ein jQuery-Objekt mit \$.myPlugin oder den jQuery-Prototyp mit \$.fn.myPlugin erweitern, fügen Sie dem \$-Namensraum nicht mehr als eine Eigenschaft hinzu. Zusätzliche öffentliche Methoden und Eigenschaften sollten dem Namensraum des Plug-ins hinzugefügt werden (z.B. \$.myPlugin.publicMethod oder \$.fn.myPlugin.pluginProperty).
- Statten Sie das Plug-in mit einer Map mit Voreinstellungen aus: \$.fn.myPlugin.defaults = {size: 'large'}.
- Ermöglichen Sie es dem Plug-in-Benutzer, alle Voreinstellungen für alle folgenden Aufrufe der Methode (\$.fn.myPlugin.defaults.size = 'medium';) oder für einen einzelnen Aufruf (\$('div').myPlugin({size: 'small'});) zu überschreiben.
- Wenn Sie den jQuery-Prototyp (\$.fn.myPlugin) erweitern, geben Sie this zurück, damit der Benutzer des Plug-ins zusätzliche jQuery-Methoden verketten kann (z.B. \$('div').myPlugin().find('p').addClass('foo')).
- Wenn Sie den jQuery-Prototyp (\$.fn.myPlugin) erweitern, erzwingen Sie die implizite Iteration durch Aufruf von this.each().
- Nutzen Sie Callback-Funktionen wo sinnvoll, um eine flexible Änderung des Plug-in-Verhaltens zu ermöglichen, ohne dessen Code zu verändern.
- Wenn das Plug-in Elemente der Benutzerschnittstelle aufruft oder den Status von Elementen überwacht, erstellen Sie es mit der Widget-Factory der jQuery UI.
- Nutzen Sie automatisierte Unit Tests für Ihre Plug-ins, wie z.B. QUnit, um sicherzustellen, dass sie wie erwartet funktionieren (mehr Informationen zu QUnit erfahren Sie in Anhang B).
- Verwenden Sie ein Versionskontrollsystem wie Git, um Änderungen am Code zu verfolgen. Überlegen Sie sich, das Plug-in auf GitHub öffentlich zu machen (<http://github.com>), damit andere etwas dazu beitragen können.
- Wenn Sie das Plug-in für andere freigeben, erklären Sie die Lizenzbedingungen genau. Denken Sie über eine MIT-Lizenz nach, die auch jQuery nutzt.

### **8.6.1 Plug-ins veröffentlichen**

Bei Beachtung der vorhergehenden Empfehlungen sollte ein Plug-in herauskommen, auf das Sie stolz sein können. Wenn es eine nützliche und häufig auftretende Aufgabe verrichtet, möchten Sie es vielleicht der jQuery-Community zur Verfügung stellen.

Neben der Aufbereitung des Plug-in-Codes wie angegeben sollten Sie auch darauf achten, die Funktionen vorher genau zu dokumentieren. Sie können die Art der Dokumentation wählen, die Ihnen liegt, oder Sie sehen sich einen Standard an wie *ScriptDoc* (<http://www.scriptdoc.org/>). Ungeachtet des Formats müssen Sie sicherstellen, dass Ihre Dokumentation jeden Parameter und jede Option der Methoden Ihres Plug-ins beschreibt.

Der Plug-in-Code und die Dokumentation können überall gehostet werden. GitHub (<http://github.com/>) ist beliebt und kostenlos. Um Ihre Arbeit öffentlich zu machen, können Sie Informationen darüber auch im offiziellen jQuery-Plug-in-Repository unter <http://plugins.jquery.com/> ablegen. Hier können Sie sich einloggen oder bei Bedarf registrieren und den Anweisungen folgen, wie Ihr Plug-in zu beschreiben ist, und angeben, wo sich der Code und die Dokumentation befinden.

## **8.7 Zusammenfassung**

In diesem Kapitel haben wir gelernt, warum die durch den jQuery-Kern bereitgestellte Funktionalität nicht die Leistungsfähigkeit der Bibliothek beschränkt. Zusätzlich zu den bereits in Kapitel 7 gezeigten, schon vorhandenen Plug-ins können wir jetzt auch eigene erstellen.

Die von uns erstellten Plug-ins enthalten verschiedene Funktionen, wie globale Funktionen, die die jQuery-Bibliothek nutzen, neue Methoden des jQuery-Objekts zur Interaktion mit DOM-Elementen und ausgereifte jQuery-UI-Widgets. Mit diesen Werkzeugen können Sie jQuery in jede gewünschte Form bringen – und Ihren JavaScript-Code auch.

## 8.8 Übungsaufgaben

Um diese Übungsaufgaben durchzuarbeiten, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) heruntergeladen werden.

Schwierige Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Erstellen Sie neue Plug-in-Methoden namens `.slideFadeIn()` und `.slideFadeOut()`, die die Deckkraftanimationen von `.fadeIn()` und `.fadeOut()` mit den Größenanimationen von `.slideDown()` und `.slideUp()` kombinieren.
2. Erweitern Sie die Anpassbarkeit der Methode `.shadow()`, sodass der Z-Index der Kopien vom Benutzer angegeben werden kann. Fügen Sie dem Quickinfo-Widget eine neue Untermethode namens `isOpen` hinzu. Diese Untermethode soll `true` zurückgeben, wenn die Quickinfo gerade angezeigt wird, ansonsten `false`.
3. Fügen Sie Code hinzu, der auf das Ereignis `tooltipopen` lauscht, das das Widget auslöst, und geben Sie auf der Konsole eine Meldung aus.
4. *Schwierig:* Bieten Sie eine alternative `content`-Option für das Quickinfo-Widget an, die den Inhalt der Seite und die Ankerpunkte via Ajax ausliest und diesen Inhalt als Quickinfo anzeigt.
5. *Schwierig:* Bieten Sie eine neue `effect`-Option für das Quickinfo-Widget an, die den Namen des jQuery-UI-Effekts (wie `explode`) auf Wunsch zusätzlich zur Anzeige bringt.

## 9 Komplexe Selektoren und Durchlaufen des DOM

Im Januar 2009 führte der Entwickler von jQuery, John Resig, ein neues Open-Source-Projekt in JavaScript namens *Sizzle* ein. Auf Sizzle, eine autarke CSS-Selektor-Engine, sollte jede JavaScript-Bibliothek mit nur geringen oder gar keinen Anpassungen an der Codebasis zugreifen können. Seit Version 1.3 hat jQuery Sizzle als Selektor-Engine quasi adoptiert.

Sizzle ist die Komponente in jQuery, die für die Verarbeitung der CSS-Selektorausdrücke innerhalb der `$()`-Funktion zuständig ist. Es legt fest, welche nativen DOM-Methoden verwendet werden, und erstellt eine Elementsammlung, auf die wir mit anderen jQuery-Methoden zugreifen können. Die Kombination aus Sizzle und den jQuery-Traversierungsmethoden ist ein extrem leistungsfähiges Werkzeug zum Durchsuchen der Seite. Es bietet einen großen Funktionsumfang, eine erweiterbare Architektur und einen ausgeklügelten Satz interner Funktionen, die eine große Leistungsfähigkeit, Flexibilität und Geschwindigkeit ermöglichen.

### 9.1 Auswahl und Durchlaufen – Teil 2

Es gibt so viele Optionen in jQuery für die Suche nach Elementen, dass wir sie nicht alle detailliert in diesem Buch besprechen können. Stattdessen haben wir uns in Kapitel 2, »*Elemente auswählen*«, die grundlegenden Arten von Selektoren und Traversierungsmethoden angesehen, sodass wir eine grobe Übersicht darüber besitzen, was es insgesamt gibt, für den Fall, dass wir noch mehr lernen müssen. Um einen genaueren Blick auf diese Abläufe zu werfen, erzeugen wir ein Skript, das uns weitere Beispiele für Auswählen und Durchlaufen bietet, die wir analysieren können.

Für unser Beispiel erzeugen wir ein HTML-Dokument, das eine Liste mit Nachrichtenmeldungen enthält. Wir platzieren diese Einträge in Tabellen, sodass wir mit der Auswahl von Zeilen und Spalten auf unterschiedliche Art und Weise experimentieren können:

```
<div id="topics">
    Topics:
    <a href="topics/all.html" class="selected">All</a>
    <a href="topics/community.html">Community</a>
    <a href="topics/conferences.html">Conferences</a>
    <!-- continues... -->
</div>
<table id="news">
    <thead>
        <tr>
            <th>Date</th> <th>Headline</th>
            <th>Author</th> <th>Topic</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th colspan="4">2011</th>
        </tr>
        <tr>
            <td>Feb 24</td> <td>jQuery Conference 2011</td>
            <td>Ralph Whitbeck</td> <td>Conferences</td>
        </tr>
        <tr>
            <td>Jan 31</td> <td>jQuery 1.5 Released</td>
            <td>John Resig</td> <td>Releases</td>
        </tr>
        <!-- continues... -->
    </tbody>
</table>
```

Anhand dieses Codefragments können wir die Dokumentstruktur erkennen. Die Tabelle besitzt vier Spalten für Datum, Überschrift, Autor und Thema, allerdings enthalten manche Zeilen nur eine »Unterüberschrift« für das Kalenderjahr. Wir müssen diese Struktur im Kopf haben, wenn wir Verbesserungen mit JavaScript an der Tabelle vornehmen:

jQuery News			
Topics: <a href="#">All</a> <a href="#">Community</a> <a href="#">Conferences</a> <a href="#">Documentation</a> <a href="#">Plugins</a> <a href="#">Releases</a> <a href="#">Miscellaneous</a>			
Date	Headline	Author	Topic
<strong>2011</strong>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	jQuery 1.5 Released	John Resig	Releases
Jan 30	API Documentation Changes	Karl Swedberg	Documentation
<strong>2010</strong>			
Nov 23	Team Spotlight: The jQuery Bug Triage Team	Paul Irish	Community
Oct 4	New Official jQuery Plugins Provide Templating, Data Linking and Globalization	John Resig	Plugins
Sep 4	The Official jQuery Podcast Has a New Home	Ralph Whitbeck	Documentation

Vor der Tabelle steht eine Reihe von Links, die die verschiedenen Themen in der Tabelle darstellen. In unserer ersten Aufgabe ändern wir das Verhalten dieser Links, sodass sie den Tabelleninhalt an Ort und Stelle filtern, statt durch die verschiedenen Seiten zu navigieren.

### 9.1.1 Dynamisches Filtern von Tabellen

Um die Thema-Links zum Filtern der Tabelle zu verwenden, müssen wir ihr normales Verhalten verhindern. Außerdem sollten wir dem Benutzer auch Rückmeldung über das momentan ausgewählte Thema bieten:

```
$(document).ready(function() {
    $('#topics a').click(function() {
        $('#topics a').removeClass('selected');
        $(this).addClass('selected');

        return false;
    });
});
```

*Listing 9–1*

Wir entfernen die Klasse `selected` aus allen Thema-Links, wenn ein Thema angeklickt wird, und fügen dann dem neuen Thema die Klasse `selected` hinzu. Die Anweisung `return false` verhindert, dass dem Link gefolgt wird.

Nun müssen wir die eigentliche Filteroperation durchführen. Als ersten Schritt verbergen wir alle Zeilen der Tabelle, die nicht das ausgewählte Thema enthalten, wie im folgenden Codefragment gezeigt:

```
// Unvollständiger Code
$(document).ready(function() {
    $('#topics a').click(function() {
        var topic = $(this).text();

        $('#news a.selected').removeClass('selected');
        $(this).addClass('selected');

        $('#news tr').show();
        if (topic != 'All') {
            $('#news tr:has(td):not(:contains("' + topic + '"))')
                .hide();
        }
        return false;
    });
});
```

*Listing 9–2*

Dazu legen wir den Text des Links in der Variablen `topic` ab, sodass wir ihn mit dem Text in der Tabelle selbst vergleichen können. Zuerst lassen wir alle Tabellenzeilen anzeigen und wenn `topic` nicht `All` enthält, verbergen wir die irrelevan-

ten Zeilen. Der für diesen Vorgang verwendete Selektor ist ein wenig komplex und bedarf der Erläuterung.

Der Selektor startet direkt und sucht mit `#news tr` die Tabellenzeilen. Wir filtern diese Elemente dann mit dem eigenen Selektor `:has()`. Dieser Selektor reduziert die momentan ausgewählten Elemente auf diejenigen, die den entsprechenden Nachkommen enthalten. In diesem Fall nehmen wir die Kopfzeilen (wie die Kalenderjahre) aus der Prüfung heraus, da sie keine `<td>`-Zellen enthalten.

Wenn wir die Zeilen mit dem eigentlichen Inhalt gefunden haben, müssen wir herausfinden, welche zum ausgewählten Thema gehören. Der eigene Selektor `:contains()` passt genau auf die Elemente, die den vorgegebenen String enthalten. Ein Verpacken in `:not()` liefert uns alle Zeilen, die den Thema-String nicht enthalten, sodass wir sie verbergen können.

Dieser Code funktioniert ganz gut, es sei denn das Thema erscheint z.B. als Teil der Nachrichtenüberschrift. Wir müssen auch die Möglichkeit berücksichtigen, dass ein Thema ein Teilstring eines anderen ist. Um diese Fälle abzudecken, müssen wir folgenden Code für jede Zeile ausführen:

```
if (topic != 'All') {  
    $('#news').find('tr:has(td)').not(function() {  
        return $(this).children(':nth-child(4)').text() == topic;  
    }).hide();  
}
```

#### **Listing 9–3**

Dieser neue Code, der den entsprechenden Abschnitt von Listing 9–2 ersetzt, tauscht einige der komplizierten Selektorausdrücke durch DOM-Traversierungsmethoden. Die Methode `.find()` funktioniert wie der Leerraum, der vorher `#news` und `tr` getrennt hat, die Methode `.not()` jedoch kann etwas leisten, wozu `:not()` nicht in der Lage ist. Wie wir bei der `.filter()`-Methode in Kapitel 2 gesehen haben, kann `.not()` eine Callback-Funktion übernehmen, die einmal je geprüftes Element aufgerufen wird. Gibt diese Funktion `true` zurück, wird das Element aus der Ergebnismenge ausgeschlossen.

#### **Selektoren und Traversierungsmethoden im Vergleich**

Die Entscheidung, einen Selektor oder die entsprechende Traversierungsmethode einzusetzen, hat auch Auswirkungen auf die Performance. Darauf werden wir später in diesem Kapitel näher eingehen.

In der Filterfunktion der `.not()`-Methode prüfen wir die Kindelemente der Zeile, um das vierte Element zu finden (das ist die Zelle in der Spalte *Topic*). Eine einfache Prüfung des Texts dieser Zelle gibt Auskunft darüber, ob die Zeile verborgen werden sollte:

Topics: All Community Conferences Documentation Plugins Releases Miscellaneous				
Date	Headline	Author	Topic	
<b>2011</b>				
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences	
<b>2010</b>				
Aug 24	jQuery Conference 2010: Boston	Ralph Whitbeck	Conferences	
Jun 14	Seattle jQuery Open Space and Hack Attack with John Resig	Rey Bango	Conferences	
Mar 15	jQuery Conference 2010: San Francisco Bay Area	Mike Hostetler	Conferences	
<b>2009</b>				
Oct 22	jQuery Summit	John Resig	Conferences	

### 9.1.2 Streifenmuster für Tabellenzeilen

In Kapitel 2 haben wir bei der Untersuchung von Selektoren herausgefunden, wie wir die Farbe von Tabellenzeilen abwechselnd ändern können. Wir haben gesehen, wie uns die eigenen Selektoren :even und :odd diese Aufgabe erleichtern und dass die CSS-Pseudoklasse :nth-child() diese Aufgabe ebenfalls erledigt:

```
$(document).ready(function() {
    $('#news').find('tr:nth-child(even)').addClass('alt');
});
```

*Listing 9–4*

Dieser einfache Selektor findet jede zweite Tabellenzeile und da die Nachrichten jedes Jahres in ihrem eigenen <tbody>-Element stehen, beginnt die Zählung mit jedem Abschnitt neu, wie im folgenden Screenshot gezeigt:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	jQuery 1.5 Released	John Resig	Releases
Jan 30	API Documentation Changes	Karl Swedberg	Documentation
<b>2010</b>			
Nov 23	Team Spotlight: The jQuery Bug Triage Team	Paul Irish	Community
Oct 4	New Official jQuery Plugins Provide Templating, Data Linking and Globalization	John Resig	Plugins

Um das Streifenmuster noch etwas komplizierter zu machen, können wir versuchen, der alt-Klasse zwei Zeilensätze gleichzeitig zu geben. Die ersten beiden Zeilen gelangen zur Klasse, die nächsten beiden nicht usw. Um das zu erreichen, müssen wir uns an die Filterfunktionen erinnern:

```
$(document).ready(function() {
    $('#news tr').filter(function(index) {
        return (index % 4) < 2;
    }).addClass('alt');
});
```

**Listing 9–5**

In unserem `.filter()`-Beispiel in Kapitel 2 und im `.not()`-Beispiel in Listing 9–3 haben unsere Filterfunktionen jedes Element (jeweils im Schlüsselwort `this`) darauf geprüft, ob es im nächsten Ergebnissatz enthalten sein soll. In unserem Beispiel wollen wir stattdessen seine Position im ursprünglichen Elementsatz wissen. Diese Information – wir wollen sie `index` nennen – wird als Argument der Funktion übergeben.

Der `index`-Parameter enthält jetzt die Position des Elements (die Zählung beginnt bei null). Damit können wir den Modulo-Operator (%) verwenden, um zu entscheiden, ob wir uns in einem Elementpaar befinden, das zur `alt`-Klasse gelangen sollte, oder nicht. Jetzt haben wir durchgehend Zweierstreifen von Zeilen in der Tabelle.

Trotzdem müssen wir noch einige Dinge erledigen. Da wir `:nth-child()` nicht mehr verwenden, beginnt die Pärchenbildung nicht bei jedem `<tbody>` neu. Außerdem sollten wir die Tabellenkopfzeilen zugunsten eines einheitlichen Aussehens weglassen. Diese Ziele werden erreicht, indem wir einige kleine Modifikationen vornehmen:

```
$(document).ready(function() {
    $('#news tbody').each(function() {
        $(this).children().has('td').filter(function(index) {
            return (index % 4) < 2;
        }).addClass('alt');
    });
});
```

**Listing 9–6**

Um jede Gruppe von Zeilen unabhängig zu behandeln, können wir mit jedem Aufruf von `.each()` eine Schleife über die `<tbody>`-Elemente laufen lassen. In dieser Schleife schließen wir die Unterüberschriften aus, wie wir es in Listing 9–3 mit `.has()` getan haben:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	jQuery 1.5 Released	John Resig	Releases
Jan 30	API Documentation Changes	Karl Swedberg	Documentation
<b>2010</b>			
Nov 23	Team Spotlight: The jQuery Bug Triage Team	Paul Irish	Community

### 9.1.3 Filter und Streifenmuster kombinieren

Unsere fortgeschrittene Tabellenbearbeitung funktioniert jetzt prima, verhält sich jedoch komisch, wenn der *Topic*-Filter verwendet wird. Damit beide Funktionen gut zusammenarbeiten, müssen wir die Tabelle jedes Mal zerlegen, wenn der Filter angewendet wird. Wir müssen außerdem berücksichtigen, welche Zeilen momentan verborgen sind, wenn wir berechnen, wo die *alt*-Klasse zum Einsatz kommt, wie im folgenden Codefragment gezeigt:

```
$(document).ready(function() {
    function stripe() {
        $('#news').find('tr.alt').removeClass('alt');
        $('#news tbody').each(function() {
            $(this).children(':visible').has('td')
                .filter(function(index) {
                    return (index % 4) < 2;
                }).addClass('alt');
        });
    }
    stripe();
    $('#topics a').click(function() {
        var topic = $(this).text();

        $('#topics a.selected').removeClass('selected');
        $(this).addClass('selected');

        $('#news').find('tr').show();
        if (topic != 'All') {
            $('#news').find('tr:has(td)').not(function() {
                return $(this).children(':nth-child(4)')
                    .text() == topic;
            }).hide();
        }
        stripe();
        return false;
    });
});
});
```

**Listing 9–7**

Dieses Skript kombiniert den Filtercode von früher mit der Zeilenzerlegung, indem es eine Funktion namens *stripe()* definiert, die einmal beim Laden des Dokuments aufgerufen wird und dann jedes Mal, wenn auf einen Thema-Link geklickt wird. In dieser Funktion achten wir darauf, die *alt*-Klasse aus nicht mehr benötigten Zeilen zu entfernen und die Auswahl der Zeilen auf die gerade angezeigten zu beschränken. Die Pseudoklasse *:visible* und ihr Gegenstück *:hidden* sind extrem nützlich und berücksichtigen, welche Elemente aus verschiedenen Gründen (u.a. ein *display*-Wert von *none* oder *width* und *height* mit einem Wert 0) verborgen sind:

Topics: All Community Conferences Documentation Plugins Releases Miscellaneous					
Date	Headline			Author	Topic
2011	Feb 24 jQuery Conference 2011: San Francisco Bay Area			Ralph Whitbeck	Conferences
2010	Aug 24 jQuery Conference 2010: Boston			Ralph Whitbeck	Conferences
	Jun 14 Seattle jQuery Open Space and Hack Attack with John Resig			Rey Bango	Conferences
	Mar 15 jQuery Conference 2010: San Francisco Bay Area			Mike Hostetler	Conferences
2009	Oct 22 jQuery Conf 2009			Mike Hostetler	Conferences

### 9.1.4 Weitere Selektoren und Traversierungsmethoden

Auch nach allen gezeigten Beispielen können wir noch lange nicht jeden Weg erklären, wie Sie mit jQuery Elemente auf einer Seite finden. Es gibt Dutzende Selektoren und DOM-Traversierungsmethoden und alle haben einen bestimmten Zweck, für den wir sie verwenden können.

Um den passenden Selektor oder die passende Methode für unsere Zwecke zu finden, gibt es viele Wege. Die Kurzreferenz am Ende dieses Buches listet jeden Selektor und jede Methode mit einer kurzen Beschreibung auf. Längere Beschreibungen und Anwendungsbeispiele finden wir jedoch an anderer Stelle, wie dem *jQuery Reference Guide*, den es in gedruckter Form oder online gibt. Sie finden dort alle Selektoren unter <http://api.jquery.com/category/selectors/> und alle Traversierungsmethoden unter <http://api.jquery.com/category/traversing/>.

## 9.2 Selektoren anpassen und optimieren

Die vielen gezeigten Techniken bieten uns eine Werkzeugkiste, mit der wir jedes gewünschte Seitenelement finden können. Damit ist die Geschichte jedoch noch nicht zu Ende. Es gibt noch viel über eine effiziente Elementsuche zu lernen. Die Effizienz kann sowohl in einem Code liegen, der leichter zu schreiben und zu lesen ist, als auch in einem Code, der schneller im Browser abgearbeitet wird.

### 9.2.1 Ein eigenes Selektor-Plug-in schreiben

Eine Verbesserung ist die Kapselung von Codefragmenten in wiederverwendbare Komponenten. Dies geschieht immer dann, wenn wir Funktionen erstellen. In Kapitel 8 haben wir diese Idee umgesetzt, indem wir jQuery-Plug-ins entwickelt haben, die Methoden in jQuery-Objekte einfügten. Das ist aber nicht der einzige Weg, wie Plug-ins bei der Wiederverwendung von Code helfen. Plug-ins können auch zusätzliche Selektorausdrücke bieten, wie der Selektor `:paused` aus Cycle in Kapitel 7.

Die einfachste Art von einem Selektorausdruck, den man hinzufügen möchte, ist eine Pseudoklasse. Dies sind die Ausdrücke, die mit einem Doppelpunkt beginnen, wie :checked oder :nth-child(). Um zu zeigen, wie ein Selektorausdruck erstellt wird, erzeugen wir eine Pseudoklasse namens :group(). Dieser neue Selektor kapselt den Code, den wir verwendet haben, um die Tabellenzeilen für das Streifenmuster in Listing 9–6 zu finden.

Wenn wir einen Selektorausdruck verwenden, um Elemente zu finden, sucht jQuery in einer internen Map namens `expr` nach Anweisungen. Die Werte in dieser Map verhalten sich wie die Filterfunktionen, die wir an `.filter()` oder `.not()` übergeben. Sie enthalten JavaScript-Code, der jedes Element in den Ergebnissatz überführt, und zwar dann und nur dann, wenn die Funktion `true` zurückgibt. Wir können dieser Map neue Ausdrücke hinzufügen, indem wir die Funktion `$.extend()` verwenden:

```
(function($) {
    $.extend($.expr[':'], {
        group: function(element, index, matches, set) {
            var num = parseInt(matches[3], 10);
            if (isNaN(num)) {
                return false;
            }
            return index % (num * 2) < num;
        }
    });
})(jQuery);
```

**Listing 9–8**

Dieser Code zeigt jQuery, dass `group` ein gültiger String ist, der in einem Selektorausdruck auf einen Doppelpunkt folgen kann, und dass die angegebene Funktion bei seinem Erscheinen aufgerufen werden soll, um festzustellen, ob das Element im Ergebnissatz enthalten sein soll.

Der hier verwendeten Funktion werden vier Parameter übergeben:

1. `element`: Das zu prüfende DOM-Element. Für die meisten Selektoren notwendig, nicht aber für unseren.
2. `index`: Der Index des DOM-Elements in unserem Ergebnissatz.
3. `matches`: Ein Array mit dem Ergebnis des regulären Ausdrucks, der zum Parsen dieses Selektors verwendet wurde. Normalerweise ist `matches [3]` das einzige relevante Element in diesem Array. In einem Selektor der Form `:a(b)` enthält `matches [3]` den Text zwischen den Klammern, also `b`.
4. `set`: Der komplette Satz von bisher passenden DOM-Elementen. Dieser Parameter wird nur selten benötigt.

Pseudoklassenselektoren müssen die in diesen vier Argumenten enthaltenen Informationen verwenden, um zu bestimmen, ob das Element in den Ergebnissatz gehört oder nicht. In diesem Fall benötigen wir nur `index` und `matches`.

Mit dem neuen Selektor `:group` haben wir einen flexiblen Weg, abwechselnde Gruppen von Elementen auszuwählen. Wir können z.B. den Selektorausdruck und die `.filter()`-Funktion aus Listing 9–5 in einem einzelnen Selektorausdruck kombinieren: `$('#news tr:group(2)')`. Alternativ können wir das abschnittsweise Verhalten aus Listing 9–7 erhalten und `:group()` als Ausdruck in einem `.filter()`-Aufruf verwenden. Wir können sogar die Anzahl der zu gruppierenden Zeilen ändern, indem wir einfach die Zahl in Klammern ändern:

```
$(document).ready(function() {
    function stripe() {
        $('#news').find('tr.alt').removeClass('alt');
        $('#news tbody').each(function() {
            $(this).children(':visible').has('td')
                .filter(':group(3)').addClass('alt');
        });
    }
    stripe();
});
```

**Listing 9–9**

In der folgenden Abbildung können wir sehen, dass sich die Zeilen in Dreiergruppen abwechseln:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	jQuery 1.5 Released	John Resig	Releases
Jan 30	API Documentation Changes	Karl Swedberg	Documentation
<b>2010</b>			
Nov 23	Team Spotlight: The jQuery Bug Triage Team	Paul Irish	Community

### 9.2.2 Selektor-Performance

Wenn wir Websites entwickeln, müssen wir die Zeit im Auge behalten, die wir für deren Erstellung benötigen, die Bequemlichkeit und Geschwindigkeit, mit der wir den Code verändern können, und die Performance der Seite bei der Interaktion mit dem Benutzer. Normalerweise sind die beiden ersten Punkte wichtiger als der dritte. Besonders bei Client-Side-Scripting können Entwickler leicht in die Fallgruben namens *Überoptimierung* und *Mikrooptimierung* stolpern und unzählige Stunden damit verbringen, Millisekunden aus der JavaScript-Ausführung herauszukitzeln. Das gilt insbesondere dann, wenn es keine nennenswerten Verzögerungen gibt und der Geschwindigkeitszuwachs für den Menschen kaum wahrnehmbar ist. Deshalb ist es ein guter Ansatz, Entwicklerzeit höher zu bewerten als Rechenzeit, es sei denn, wir bemerken, dass die Anwendung langsam wird.

Auch wenn die Geschwindigkeit eine Rolle spielt, kann es schwierig sein, die Engpässe in unserem jQuery-Code zu finden. Wir haben bereits früher darauf hingewiesen, dass einige Selektoren generell schneller als andere sind. Solche Selektoren in Traversierungsmethoden zu verlagern, kann dabei helfen, die Suchdauer für Seitenelemente zu verkürzen. Die Performance von Selektoren und DOM-Methoden ist daher ein guter Ansatz für die Suche nach gewissen Verzögerungen in unserem Code, die der Benutzer bemerken kann. Die Aussagen über die Performance von Selektoren und Methoden sind jedoch überholt, wenn neue, schnellere Browser auf den Markt kommen und neue jQuery-Versionen Geschwindigkeitsverbesserungen bringen. Mit diesen Überlegungen im Hinterkopf sehen wir uns einige Richtlinien an und vergessen dabei nicht, dass nichts so bleibt, wie es ist, und dass auch die grundlegenden Annahmen überholt sein können, wenn es um neue Browser, jQuery-Versionen und Webseiten geht.

### Implementierung des Sizzle-Selektors

Wie wir zu Beginn des Kapitels festgestellt haben, parst die jQuery-Implementierung von Sizzle einen Ausdruck, wenn wir der `$()`-Funktion einen Selektorausdruck übergeben, und entscheidet, wie die enthaltenen Elemente gesammelt werden sollen. In der Grundform setzt Sizzle die effizienteste *DOM-Methode* ein, die der Browser unterstützt, um eine *Knotenliste* zu erhalten. Es handelt sich dabei um ein natives, Array-ähnliches Objekt mit DOM-Elementen, die schließlich von jQuery in ein richtiges Array konvertiert und dem Objekt *jQuery* hinzugefügt werden. Nachfolgend ist eine Liste von DOM-Methoden aufgeführt, die jQuery intern verwendet, und den frühesten Browsersversionen, die sie unterstützen:

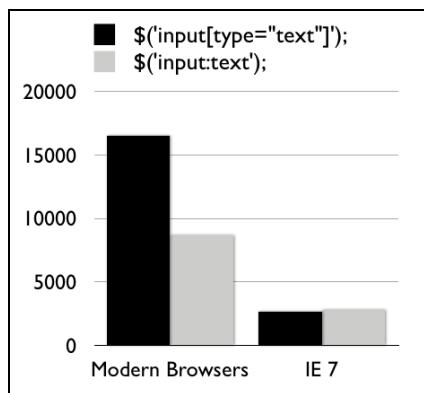
- `.getElementById()` wählt ein eindeutiges Element mit einer ID aus, die einem gegebenen String entspricht. Unterstützt von IE 6+, Firefox 3+, Safari 3+, Chrome 4+ und Opera 10+.
- `.getElementsByTagName()` wählt alle Elemente mit einem Tag-Namen, der einem gegebenen String entspricht. Unterstützt von IE 6+, Firefox 3+, Safari 3+, Chrome 4+ und Opera 10+.
- `.getElementsByClassName()` wählt alle Elemente, von denen ein Klassenname dem übergebenen String entspricht. Unterstützt von IE 9+, Firefox 3+, Safari 4+, Chrome 4+ und Opera 10+.
- `.querySelectorAll()` wählt alle Elemente, die einem übergebenen Selektorausdruck entsprechen. Unterstützt von IE 8+, Firefox 3.5+, Safari 3+, Chrome 4+ und Opera 10+.

Wenn ein Teil eines Selektorausdrucks nicht von einer dieser Methoden bearbeitet werden kann, geht Sizzle in einen Schleifenmodus über, in dem alle bereits gesammelten Elemente an dem Teilausdruck geprüft werden. Wenn kein Teil des Selektorausdrucks durch eine DOM-Methode bearbeitet werden kann, beginnt Sizzle mit der Sammlung aller Elemente im Dokument, dargestellt durch `document.getElementsByTagName('*')`, und durchläuft sie alle.

Die Schleife und der Test jedes einzelnen Elements sind wesentlich aufwendiger in punkto Performance als jede native DOM-Methode. Glücklicherweise enthalten die meisten aktuellen Versionen der Browser die native Methode `.querySelectorAll()`, die auch Sizzle nutzt, wenn es die andere, schnellere native Methode nicht verwenden kann – mit einer Ausnahme. Wenn der Selektorausdruck einen eigenen jQuery-Selektor enthält, wie `:eq()`, `:odd` oder `:even`, die keine CSS-Entsprechung haben, bleibt Sizzle nichts anderes übrig, als Schleifen zu durchlaufen und zu prüfen.

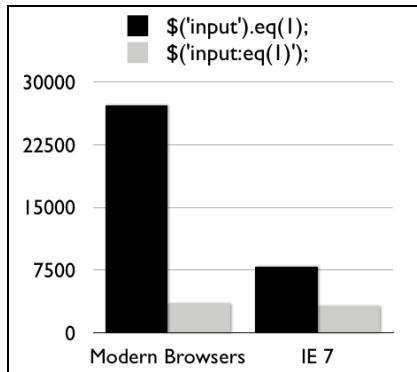
### Testen der Selektorgeschwindigkeit

Um einen Eindruck der unterschiedlichen Performance zwischen `.querySelectorAll()` und der Schleifenprüfung zu bekommen, stellen wir uns ein Dokument vor, in dem wir alle `<input type="text">`-Elemente auswählen möchten. Wir können den Selektorausdruck auf zwei Arten formulieren: `$('#input[type="text"]')` verwendet den CSS-Attributselektor und `$('#input:text')` verwendet den eigenen jQuery-Selektor. Laufen diese beiden Ausdrücke auf der JavaScript-Benchmarking-Seite <http://jsperf.com/> gegeneinander, sind die Ergebnisse sehr unterschiedlich, wie das folgende Diagramm zeigt:



In den Tests von jsperf.com wird jeder Lauf wiederholt, um zu ermitteln, wie oft er in einem bestimmten Zeitraum ablaufen kann. Je höher der Wert, desto besser. Wenn der Test in modernen Browsern erfolgt, die `.querySelectorAll()` unterstützen (Chrome 12, Firefox 4 und Safari 5), kann der Selektor Nutzen daraus ziehen und ist im Durchschnitt ungefähr zweimal so schnell wie der eigene jQuery-Selektor. Im Internet Explorer 7 verhalten sich beide Selektoren aber beinahe gleich, da beide jQuery zwingen, jedes `<input>`-Element einer Schleifenprüfung zu unterziehen.

Der Unterschied in der Performance fällt zwischen `($('#input:eq(1)')` und `($('#input').eq(1))` sogar noch größer aus:



Sogar im Internet Explorer 7 bringt die Verlagerung des eigenen Selektors `:eq()` hin zur Methode `.eq()` eine Geschwindigkeitssteigerung um mehr als 100%. Der Grund? Immer, wenn jQuery eine einzelne ID, einen Tag-Namen oder einen Klassennamen sieht, wird der Selektor zur DOM-Methode geleitet (wenn der Browser diese unterstützt). In diesem Fall führt die Verwendung des einfachen Tag-Namens `input` als Argument für die `$()`-Funktion zu einer sehr schnellen Suche. Die Methode `.eq()` ruft dann einfach eine Array-Funktion auf, um das zweite Element in der jQuery-Sammlung abzurufen.

Als Faustregel sollten wir Selektoren bevorzugen, die Teil der CSS-Spezifikation sind, und keine eigenen jQuery-Selektoren. Bevor wir jedoch unsere Selektoren modifizieren, sollten wir prüfen, ob es wirklich nötig ist, die Geschwindigkeit zu steigern. Danach sollten wir mit einem Benchmark-Tool wie <http://jsperf.com> ermitteln, wie hoch die Steigerung tatsächlich ausfällt.

### 9.3 Durchlaufen des DOM – Hinter den Kulissen

In Kapitel 2 und dann wieder zu Beginn dieses Kapitels haben wir uns angesehen, wie wir von einem Satz DOM-Elemente in einen anderen gelangen konnten, indem wir DOM-Traversierungsmethoden verwendet haben. Unsere (nicht sehr ausführliche) Übersicht solcher Methoden enthielt einfache Ansätze zum Erreichen von Nachbarzellen, wie `.next` und `.parent`, und komplexere Verfahren, wie das Kombinieren von Selektorausdrücken, wie `.find()` und `.filter()`. Inzwischen sollten wir eine ziemlich klare Vorstellung davon haben, wie wir schrittweise von einem DOM-Element zum nächsten kommen.

Jedes Mal, wenn wir einen solchen Schritt machen, merkt sich jQuery diesen Vorgang und legt eine Spur, auf der wir wieder zurückfinden können. Einige dieser Methoden, die wir in diesem Kapitel kurz streifen werden, nämlich `.end()` und `.andSelf()`, machen sich diese Protokollierung zunutze. Um so viel wie möglich

aus diesen Methoden herauszuholen und um überhaupt effizienten jQuery-Code zu schreiben, müssen wir mehr darüber lernen, wie die DOM-Traversierungsmethoden funktionieren.

### 9.3.1 jQuery-Objekteigenschaften

Wie Sie wissen, konstruieren wir eine jQuery-Objektinstanz normalerweise, indem wir einen Selektorausdruck an die `\$()`-Funktion übergeben. Im resultierenden Objekt liegt eine Array-Struktur, die Referenzierungen auf jedes DOM-Element enthält, das zum Selektor passt. Was wir nicht gesehen haben, sind die anderen im Objekt versteckten Eigenschaften. Zu diesen Eigenschaften gehören `.context`, ein Bezug auf den DOM-Knoten, an dem die Suche begann (normalerweise `document`), und `.selector`, eine Aufzeichnung des Selektorausdrucks, der dieses Objekt erzeugt hat. Diese beiden Eigenschaften spielen eine Rolle, wenn Ereignisdelegationsmethoden wie `.live()` aufgerufen werden (mehr dazu erfahren Sie in Kapitel 10). Wenn eine DOM-Methode aufgerufen wird, kommt eine dritte Eigenschaft ins Spiel: `.prevObject` enthält einen Bezug zu dem jQuery-Objekt, von dem die DOM-Methode aufgerufen wurde.

Um den Vorgang in Aktion zu sehen, können wir eine Zelle in unserer Tabelle hervorheben und die Eigenschaften untersuchen:

```
$(document).ready(function() {  
    var $cell = $('#release');  
  
    $cell.addClass('highlight');  
    console.log($cell.context);  
    console.log($cell.selector);  
    console.log($cell.prevObject);  
});
```

**Listing 9–10**

Das vorhergehende Codefragment hebt die einzelne ausgewählte Zelle hervor, wie der folgende Screenshot zeigt:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	<b>jQuery 1.5 Released</b>	John Resig	Releases
Jan 30	API Documentation Changes	Karl Swedberg	Documentation

Auf der Konsole werden drei Meldungen ausgegeben:

Ausdruck	Gespeicherter Wert
\$cell.context	Document
\$cell.selector	#release
\$cell.prevObject	undefined

Wir sehen, dass .context das Dokumentobjekt ist, .selector genau der String, den wir dem Objekt übergeben haben, und .prevObject ist nicht definiert, da es sich um ein neu erzeugtes Objekt handelt. Wenn wir dem Mix eine DOM-Traversierungsmethode hinzufügen, wird die Sache interessanter:

```
$(document).ready(function() {
    var $cell = $('#release').nextAll();

    $cell.addClass('highlight');
    console.log($cell.context);
    console.log($cell.selector);
    console.log($cell.prevObject);
});
```

**Listing 9–11**

Dieses Codefragment verändert, welche Zellen hervorgehoben werden:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	jQuery 1.5 Released	<b>John Resig</b>	<b>Releases</b>
Jan 30	API Documentation Changes	Karl Swedberg	Documentation

Die Aufzeichnungen im Log ändern sich wegen des Aufrufs von .nextAll():

Ausdruck	Gespeicherter Wert
\$cell.context	Document
\$cell.selector	#release.nextAll()
\$cell.prevObject	[td]

Jetzt sind die beiden Zellen nach der ursprünglich ausgewählten hervorgehoben. Innerhalb des jQuery-Objekts zeigt .context noch immer auf das Dokumentobjekt, aber .selector wurde verändert, um unseren Aufruf von .nextAll() anzuzeigen, und .prevObject bezieht sich auf die ursprüngliche jQuery-Objektinstanz vor dem Aufruf von .nextAll().

### 9.3.2 Der DOM-Elementstack

Da jede jQuery-Objektinstanz eine `.prevObject`-Eigenschaft besitzt, die auf die vorhergehende zeigt, haben wir eine verknüpfte Listenstruktur, die einen Stack implementiert. Jeder DOM-Methodenaufruf findet einen neuen Satz Elemente und legt diesen Satz auf den Stack. Das ist nur dann von Nutzen, wenn wir etwas mit diesem Stack anstellen können, und jetzt kommen die Methoden `.end()` und `.andSelf()` ins Spiel.

Die `.end`-Methode entfernt einfach ein Element vom Ende des Stapels, was dem Wert der Eigenschaft `.prevObject` entspricht. Wir haben ein Beispiel davon in Kapitel 2 gesehen und weitere folgen in diesem Kapitel. Als interessanteres Beispiel untersuchen wir, wie `.andSelf()` den Stack manipuliert:

```
$(document).ready(function() {
    $('#release').nextAll().andSelf().addClass('highlight');
});
```

**Listing 9-12**

Wieder haben sich die hervorgehobenen Zellen verändert, wie der folgende Screenshot zeigt:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	<b>jQuery 1.5 Released</b>	<b>John Resig</b>	<b>Releases</b>
Jan 30	API Documentation Changes	Karl Swedberg	Documentation

Wenn `.andSelf()` aufgerufen wird, sieht jQuery einen Schritt weiter hinten auf dem Stack nach und kombiniert die beiden Elementsätze. In unserem Beispiel bedeutet das, dass die hervorgehobenen Zellen sowohl die vom `.nextAll()`-Aufruf gefundenen Zellen enthalten als auch die mit dem Selektor gefundene Originalzelle. Dieses neue, zusammengesetzte Element wird dann auf den Stack geschoben.

Diese Art der Stack-Manipulation kommt uns gelegen. Um sicherzustellen, dass diese Verfahren funktionieren, wenn sie benötigt werden, muss jede Implementation einer DOM-Traversierungsmethode den Stack korrekt aktualisieren. Das bedeutet, dass wir einige der internen Systemabläufe verstehen müssen, wenn wir eigene DOM-Traversierungsmethoden entwickeln wollen.

### 9.3.3 Ein Plug-in für DOM-Traversierungsmethoden schreiben

Wie jede andere jQuery-Objektmethode können jQuery-Traversierungsmethoden durch das Hinzufügen von Eigenschaften zu `$.fn` erzeugt werden. Wir haben in Kapitel 8 gesehen, dass neue von uns definierte jQuery-Methoden auf einem passenden Elementsatz ausgeführt werden und das jQuery-Objekt zurückgeben sollten, sodass Benutzer die Methoden verketten können. Wenn wir DOM-Traversierungsmethoden erzeugen, ist der Vorgang genauso, nur dass das zurückgegebene jQuery-Objekt auf einen neuen Satz passender Elemente zeigen muss.

Als Beispiel erzeugen wir ein Plug-in, das alle Tabellenzellen in derselben Spalte wie die gegebene Zelle findet. Zuerst sehen wir uns den Plug-in-Code komplett an und dann untersuchen wir ihn stückweise, um zu verstehen, wie er funktioniert:

```
(function($) {
    $.fn.column = function() {
        var $cells = $();
        this.each(function() {
            var $td = $(this).closest('td, th');
            if ($td.length) {
                var colNum = $td[0].cellIndex + 1;
                var $columnCells = $td
                    .closest('table')
                    .find('td, th')
                    .filter(':nth-child(' + colNum + ')');
                $cells = $cells.add($columnCells);
            }
        });
        return this.pushStack($cells);
    };
})(jQuery);
```

**Listing 9-13**

Unsere `.column()`-Methode könnte von einem jQuery-Objekt aufgerufen werden, das auf null, eines oder mehrere DOM-Elemente zeigt. Um alle diese Möglichkeiten zu berücksichtigen, verwenden wir die `.each()`-Methode für eine Schleife über die Elemente und fügen die Spalten der Zellen nacheinander in die Variable `$cells` ein. `$cells` beginnt als leeres jQuery-Objekt und wird dann durch die Methode `.add()` erweitert, um je nach Bedarf auf mehr und mehr DOM-Elemente zu zeigen.

Das erklärt die äußere Schleife der Funktion. Innerhalb der Schleife müssen wir verstehen, wie `$columnCells` mit den DOM-Elementen in der Tabellenspalte gefüllt wird.

Zuerst erstellen wir eine Referenz zur untersuchten Tabellenzelle. Wir wollen der `.column()`-Methode erlauben, für Tabellenzellen oder Elemente innerhalb der Tabellenzellen aufgerufen zu werden. Das erledigt die `.closest()`-Methode für uns. Sie geht im DOM-Baum nach oben, bis sie ein zu unserem Selektor passendes Element findet. Diese Methode wird außerdem sehr nützlich werden, wenn es um Ereignisdelegation geht, auf die wir in Kapitel 10 wieder treffen.

Für die Tabellenzelle ermitteln wir die Spaltennummer über die DOM-Eigenschaft `cellIndex`. Damit erhalten wir einen nullbasierten Index der Spalte. Wir addieren 1 zu diesem Wert und verwenden ihn später in einem auf eins basierenden Kontext. Von der Zelle aus bewegen wir uns zum nächsten `<table>`-Element, gehen zurück zu den Elementen `<td>` und `<th>` und filtern diese Zellen bis zur passenden Spalte mit einem `:nth-child()`-Selektorausdruck.

### Umgang mit geschachtelten Tabellen

Das Plug-in, das wir schreiben, ist durch den Aufruf `.find('td, th')` auf einfache, ungeschachtelte Tabellen beschränkt. Um geschachtelte Tabellen zu unterstützen, müssten wir feststellen, ob `<tbody>`-Tags vorhanden sind, und uns im DOM-Baum um den entsprechenden Betrag auf- und abwärts bewegen. Dadurch würde die Komplexität dieses Beispiels jedoch unangemessen steigen.

Nachdem wir alle Zellen in der oder den Spalten gefunden haben, müssen wir das neue jQuery-Objekt zurückgeben. Wir könnten einfach `$cells` aus unserer Methode zurückliefern, aber das würde den DOM-Elementstack nicht berücksichtigen. Stattdessen reichen wir `$cells` an die Methode `.pushStack()` weiter und geben das Ergebnis zurück. Diese Methode übernimmt ein Array mit DOM-Elementen und fügt sie dem Stack hinzu, sodass spätere Aufrufe von Methoden wie `.andSelf()` und `.end()` richtig funktionieren.

Um unser Plug-in in Aktion zu sehen, können wir wie im Folgenden gezeigt auf Mausklicks auf Zellen reagieren und die entsprechende Spalte hervorheben:

```
$(document).ready(function() {
    $('#news td').click(function() {
        $('#news td.active').removeClass('active');
        $(this).column().addClass('active');
    });
});
```

**Listing 9–14**

Die Klasse `active` wird der ausgewählten Spalte hinzugefügt, was zu einer anderen Einfärbung führt, wenn z.B. auf den Namen des Autors geklickt wird, wie der folgende Screenshot zeigt:

Date	Headline	Author	Topic
<b>2011</b>			
Apr 15	jQuery 1.6 Beta 1 Released	John Resig	Releases
Feb 24	jQuery Conference 2011: San Francisco Bay Area	Ralph Whitbeck	Conferences
Feb 7	New Releases, Videos & a Sneak Peek at the jQuery UI Grid	Addy Osmani	Plugins
Jan 31	<b>jQuery 1.5 Released</b>	<b>John Resig</b>	<b>Releases</b>
Jan 30	API Documentation Changes	Karl Swedberg	Documentation

### 9.3.4 Performance von DOM-Traversierungsmethoden

Die im vorangegangenen Abschnitt genannten Überlegungen zur Performance von Selektoren gelten gleichermaßen für die Performance von DOM-Traversierungsmethoden. Wir sollten wenn möglich die Priorität auf einfachen Code und gute Wartbarkeit richten und eine Optimierung nur dann betreiben, wenn die Leistung zu einem messbaren Problem wird. Auch hier helfen uns Seiten wie <http://jsperf.com> dabei, den besten Lösungsansatz zu entwickeln.

Zusätzlich zu diesem allgemeinen Rat sollten wir daran denken, das Wiederholen von Selektoren und DOM-Methoden zu minimieren. Da diese Aufrufe potenziell aufwendig sind, sollten wir sie so wenig wie möglich verwenden. Zwei hierfür sinnvolle Strategien sind die Verkettung und das Objekt-Caching.

#### Performance-Verbesserung durch Verkettung

Wir haben die Verkettung schon mehrfach behandelt, insbesondere in Bezug auf die Länge unseres Codes. Die Verkettung kann aber auch Vorteile bei der Performance bringen, indem sie Wiederholungen reduziert.

Unsere Funktion `stripe()` aus Listing 9–9 hat das Element mit der ID `news` zweimal lokalisiert: einmal, um die Klasse `alt` von Zeilen zu entfernen, die sie nicht mehr benötigen, und einmal, um die Klasse einem neuen Satz Zeilen zuzuweisen. Durch Verkettung können wir diese beiden Vorgänge in einem kombinieren und die Wiederholung sparen:

```
$(document).ready(function() {
    function stripe() {
        $('#news')
            .find('tr.alt').removeClass('alt').end()
            .find('tbody').each(function() {
                $(this).children(':visible').has('td')
                    .filter(':group(3)').addClass('alt');
            });
    }
    stripe();
});
```

*Listing 9–15*

Um die beiden Verwendungen von `$('#news')` zusammenzuführen, können wir den DOM-Elementstack innerhalb des jQuery-Objekts nutzen. Der erste Aufruf von `.find()` schiebt die Tabellenzeilen auf den Stack, `.end()` entfernt sie jedoch wieder, sodass der nächste Aufruf von `.find()` wieder auf der news-Tabelle operiert. Diese schlaue Stack-Manipulation ist eine gute Möglichkeit, doppelte Selektoren zu vermeiden.

### Performance-Verbesserung durch Caching

Caching bedeutet einfach, das Ergebnis einer Operation zu speichern, sodass es mehrfach ohne erneute Operation genutzt werden kann. Im Kontext von Selektor- und DOM-Methoden-Performance können wir Caching einsetzen, indem wir ein jQuery-Objekt zur späteren Verwendung in einer Variablen speichern, statt ein neues zu erzeugen.

Auf unser Beispiel bezogen, können wir die Funktion `stripe()` umschreiben, um doppelte Selektoren zu vermeiden, indem wir statt Verkettung Caching verwenden:

```
$(document).ready(function() {
    var $news = $('#news');
    function stripe() {
        $news.find('tr.alt').removeClass('alt');
        $news.find('tbody').each(function() {
            $(this).children(':visible').has('td')
                .filter(':group(3)').addClass('alt');
        });
    }
    stripe();
});
```

#### **Listing 9–16**

Die beiden Operationen sind wieder getrennte JavaScript-Anweisungen und nicht miteinander verkettet. Wir führen den Selektor `$('#news')` trotzdem nur einmal aus, indem wir das Ergebnis in `$news` speichern.

Dieser Ansatz ist ein wenig umfangreicher als die Verkettung, da wir die Variable, die das jQuery-Objekt aufnimmt, separat erzeugen müssen. Auf der anderen Seite hat er den Vorteil, dass zwei Verwendungen der ausgewählten Elemente im Code auch weit auseinander liegen können, wenn das sein muss. Zusätzlich können wir die ausgewählten Elemente außerhalb der `.stripe()`-Funktion speichern, sodass der Selektor nicht für jeden Funktionsaufruf neu ausgeführt werden muss.

Da die Auswahl von Seitenelementen per ID extrem schnell ist, haben beide Beispiele keine großen Auswirkungen auf die Performance und in der Praxis würden wir den am besten lesbaren und wartbaren Ansatz verfolgen. Trotzdem können diese Techniken wertvoll sein, wenn die Performance ein Problem darstellt.

## 9.4 Zusammenfassung

In diesem Kapitel sind wir tiefer in die jQuery-Funktionen zum Suchen von Seiten-elementen eingestiegen. Wir haben uns genauer angesehen, wie die Sizzle-Engine funktioniert und welche Auswirkungen sie auf das Schreiben von effektivem und effizientem Code hat. Zusätzlich haben wir die Art und Weise untersucht, wie wir jQuery-Selektoren und DOM-Traversierungsmethoden erweitern und verbessern können.

### 9.4.1 Literatur

Eine vollständige Liste von Selektoren und DOM-Traversierungsmethoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* oder in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com>.

## 9.5 Übungsaufgaben

Um die folgenden Übungsaufgaben durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Verändern Sie die Tabellenzeilen-Routine so, dass sie der ersten Zeile keine Klasse zuweist, der zweiten die Klasse `alt` und der dritten die Klasse `alt-2`. Wiederholen Sie dies für jeden Dreiersatz innerhalb eines Abschnitts.
2. Erzeugen Sie ein neues Selektor-Plug-in namens `:containsExactly()`, das Elemente mit einem Textinhalt auswählt, der genau der Vorgabe zwischen den Klammern entspricht.
3. Verwenden Sie diesen neuen `:containsExactly()`-Selektor, um den Filtercode aus Listing 9–3 umzuschreiben.
4. Erzeugen Sie ein neues DOM-Traversierungsmethoden-Plug-in namens `.grandparent()`, das sich von einem oder mehreren Elementen hin zu seinen Großeltern-Elementen im DOM bewegt.
5. *Schwierig:* Stellen Sie den Inhalt von `index.html` in die Benchmarking-Seite <http://jsperf.com> und vergleichen Sie die Leistung beim Suchen nach dem nächsten verwandten Tabellenelement von `<td id="release">` unter Verwendung von:
  - Methode `.closest()`
  - Methode `.parents()`, wobei das Ergebnis auf die erste gefundene Tabelle beschränkt ist.

6. *Schwierig:* Stellen Sie den Inhalt von `index.html` in die Benchmarking-Seite <http://jsperf.com> und vergleichen Sie die Performance beim Suchen nach dem letzten `<td>`-Element in jeder Zeile unter Verwendung von:

- Pseudoklasse `:last-child`
- Pseudoklasse `:nth-child()`
- Methode `.last()` innerhalb jeder Zeile (verwenden Sie `.each()` für eine Schleife über die Zeilen)
- Pseudoklasse `:last` innerhalb jeder Zeile (verwenden Sie `.each()` für eine Schleife über die Zeilen)

# 10 Komplexe Ereignisse

Unsere Anwendungen sind nur dann interaktiv, wenn wir die Aktionen des Benutzers beobachten und darauf reagieren. In Kapitel 3, »*Ereignisbehandlung*«, sind wir auf einige der jQuery-Funktionen für die Ereignisbehandlung eingegangen. In den darauf folgenden Kapiteln haben wir Ereignisse vielfach genutzt. Trotzdem gibt es noch viel zu entdecken.

In diesem Kapitel befassen wir uns näher mit der Ereignisdelegation und den Schwierigkeiten, die damit verbunden sind. Wir werden den negativen Einfluss bestimmter Ereignisse auf die Performance beleuchten und ein Verfahren kennenlernen, den Browser während dieser Vorgänge antwortbereit zu halten. Wir kehren wieder zu benutzerdefinierten Ereignissen zurück und nutzen dafür das spezielle Ereignissystem, das jQuery intern einsetzt.

## 10.1 Ereignisse – Teil 2

Für unser Beispieldokument erzeugen wir eine einfache Fotogalerie. Die Galerie zeigt eine Reihe von Fotos mit der Option auf weitere Fotos nach Klick auf einen Link. Wir werden das jQuery-Ereignissystem verwenden, um Textinformationen über die Fotos einzublenden, wenn der Benutzer mit der Maus darüberfährt:

```
<div id="container">
    <h1>Photo Gallery</h1>

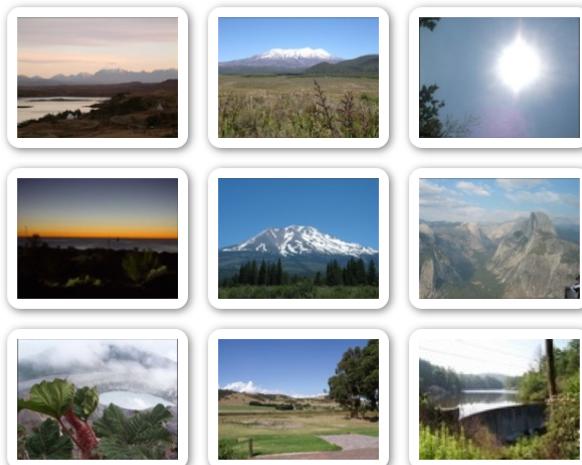
    <div id="gallery">
        <div class="photo">
            
            <div class="details">
                <div class="description">The Cuillin Mountains,
                    Isle of Skye, Scotland.</div>
                <div class="date">12/24/2000</div>
                <div class="photographer">Alasdair Dougall</div>
            </div>
        </div>
        <div class="photo">
            
```

```
<div class="details">
  <div class="description">Mt. Ruapehu in summer</div>
  <div class="date">01/13/2005</div>
  <div class="photographer">Andrew McMillan</div>
</div>
</div>
<div class="photo">
  
  <div class="details">
    <div class="description">midday sun</div>
    <div class="date">04/26/2011</div>
    <div class="photographer">Jaycee Barratt</div>
  </div>
</div>
<!-- Code geht weiter -->
</div>
<a id="more-photos" href="pages/1.html">More Photos</a>
</div>
```

### Codebeispiele zum Herunterladen

Wie viele der HTML-, CSS- und JavaScript-Beispiele dieses Buches ist der oben stehende Code nur ein Auszug des vollständigen Dokuments. Um mit den Beispielen zu experimentieren, können Sie sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen. Zusätzlich lassen sich die Beispiele in einem interaktiven Browser unter <http://book.learningjquery.com/> betrachten.

Wenn wir den Fotos Formatvorlagen zuweisen, um sie in Dreierreihen anzurichten, sieht die Galerie ähnlich aus wie folgender Screenshot.



[More Photos](#)

### 10.1.1 Zusätzliche Datenseiten laden

Mittlerweile sind wir Experten für die Standardaufgabe, auf einen Klick auf ein Seitenelement zu reagieren. Wird der Link *More Photos* angeklickt, müssen wir einen Ajax-Aufruf für die nächste Reihe Fotos starten und sie an <div id="gallery"> anfügen. Wir müssen außerdem das Ziel für den Link anpassen, um auf die nächste Reihe von Fotos zu verweisen:

```
$(document).ready(function() {
    var pageNum = 1;
    $('#more-photos').click(function() {
        var $link = $(this);
        var url = $link.attr('href');
        if (url) {
            $.get(url, function(data) {
                $('#gallery').append(data);
            });
            pageNum++;
            if (pageNum < 20) {
                $link.attr('href', 'pages/' + pageNum + '.html');
            }
            else {
                $link.remove();
            }
        }
        return false;
    });
});
```

**Listing 10–1**

Unser .click()-Handler verwendet die Variable `pageNum`, um die nächste anzufordernde Fotoseite zu verfolgen, und erstellt daraus den neuen `href`-Wert für den Link. Da die Variable `pageNum` außerhalb der Funktion definiert wird, besteht ihr Wert zwischen den Klicks auf den Link weiter. Wenn wir die letzte Seite mit Fotos erreicht haben, entfernen wir den Link.

#### Fortschreitende Verbesserung

Unser Beispiel ist so konstruiert, dass es offline funktioniert – ohne Webserver. In einer echten Implementierung würden die Daten vermutlich eher aus einer Datenbank stammen. Mithilfe von serverseitigem Code würden wir eine komplette HTML-Seite liefern, wenn der Browser eine Reihe Fotos anfordert, und nur das HTML-Fragment mit dem Foto-Markup, wenn eine Ajax-Anfrage kommt. So haben wir eine benutzerfreundliche Schnittstelle, die auch ohne JavaScript funktioniert.

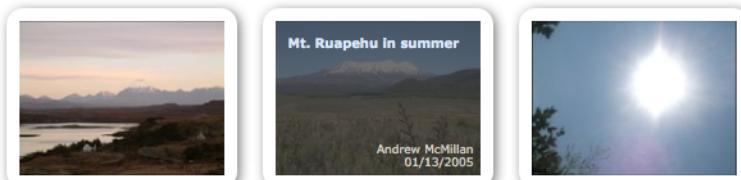
### 10.1.2 Daten beim Darüberfahren mit der Maus anzeigen

Als Nächstes müssen wir die Informationen zum Foto anzeigen, sobald sich die Maus des Benutzers in diesem Bereich der Seite befindet. Für den ersten Schritt können wir zum Anzeigen der Daten die `.hover()`-Methode wie folgt verwenden:

```
$(document).ready(function() {
    $('div.photo').hover(function() {
        $(this).find('.details').fadeTo('fast', 0.7);
    }, function() {
        $(this).find('.details').fadeOut('fast');
    });
});
```

**Listing 10-2**

Wenn die Maus ein Foto berührt, werden die Informationen mit 70% Deckkraft eingeblendet und beim Verlassen ausgeblendet:



Wie immer gibt es mehrere Wege, diese Aufgabe zu lösen. Da ein Teil jedes Handlers immer gleich ist, lassen sich die beiden kombinieren, um die Duplizierung von Code zu reduzieren. Wir können einen Handler gleichzeitig an `mouseenter` und `mouseleave` binden, indem wir die Ereignisnamen durch Leerzeichen trennen:

```
$(document).ready(function() {
    $('div.photo')
        .bind('mouseenter mouseleave', function(event) {
            var $details = $(this).find('.details');
            if (event.type == 'mouseenter') {
                $details.fadeTo('fast', 0.7);
            } else {
                $details.fadeOut('fast');
            }
        });
});
```

**Listing 10-3**

Mit dem an zwei Ereignisse gebundenen Handler prüfen wir den Ereignistyp, um festzulegen, ob die Informationen ein- oder ausgeblendet werden sollen. Der Code, der das `<div>`-Element lokalisiert, ist für beide Ereignisse gleich, sodass wir ihn nur einmal schreiben müssen.

Dieses Beispiel ist zugegebenermaßen ein wenig überzogen, da der gemeinsam genutzte Code hier so kurz ist. In anderen Fällen kann dieses Verfahren die Komplexität des Codes jedoch deutlich reduzieren. Hätten wir beispielsweise beschlossen, `mouseenter` eine Klasse hinzuzufügen und die von `mouseleave` zu entfernen, statt die Deckkraft zu variieren, hätten wir das mit einer einzigen Anweisung im Handler erledigt:

```
$(this).find('.details')
    .toggleClass('entered', event.type == 'mouseenter');
```

Auf jeden Fall funktioniert unser Skript jetzt wie gewünscht, abgesehen davon, dass wir die zusätzlich zu ladenden Fotos noch nicht berücksichtigt haben, wenn der Benutzer auf den Link MORE PHOTOS klickt. Wie in Kapitel 3 bemerkt, sind Ereignishandler nur an Elemente gebunden, die zum Zeitpunkt des `.bind()`-Aufrufs vorhanden sind. Später hinzugefügte Elemente, z.B. aus einem Ajax-Aufruf, teilen dieses Verhalten nicht. Wir haben gesehen, dass zwei Ansätze, dieses Problem zu lösen, darin bestehen, den Ereignishandler neu zu binden, wenn neuer Inhalt vorhanden ist, oder die Handler zu Anfang an ein Containerelement zu binden und sich auf das *Event Bubbling* zu verlassen. Wir werden hier den zweiten Ansatz, die *Ereignisdelegation*, verfolgen.

## 10.2 Ereignisdelegation

Um Ereignisdelegation zu implementieren, müssen wir die `target`-Eigenschaft des Objekts `event` darauf prüfen, ob sie mit dem Element übereinstimmt, das das Verhalten auslösen soll. Das Ereignisziel stellt das innerste, am tiefsten verschachtelte Element dar, das das Ereignis empfängt. In unserem Beispiel-HTML stehen wir diesmal vor einer neuen Herausforderung. Die `<div class="photo">`-Elemente sind höchstwahrscheinlich nicht Ziel für das Ereignis, da sie andere Elemente enthalten, wie das Bild und dessen Informationen. Um dieses `<div>` in einem Element zu finden, das es enthält, können wir die Methode `.closest()` verwenden, die sich im DOM von Elternelement zu Elternelement hocharbeitet, bis sie ein Element findet, das dem Selektorausdruck entspricht. Werden keine Elemente gefunden, funktioniert sie wie jede andere DOM-Traversierungsmethode und gibt ein neues, »leeres« jQuery-Objekt zurück, so wie hier:

```
// Unfinished code
$(document).ready(function() {
    $('#gallery').bind('mouseover mouseout', function(event) {
        var $target = $(event.target).closest('div.photo');
        var $details = $target.find('.details');
        var $related = $(event.relatedTarget)
            .closest('div.photo');
```

```
if ( event.type == 'mouseover' && $target.length ) {  
    $details.fadeTo('fast', 0.7);  
} else if (event.type == 'mouseout' && !$related.length) {  
    $details.fadeOut('fast');  
}  
});  
});
```

**Listing 10–4**

Beachten Sie, dass wir auch den Ereignistyp von mouseenter und mouseleave auf mouseover und mouseout umstellen müssen, da die vorherigen Typen nur ausgelöst werden, wenn die Maus erstmalig das `<div>`-Element der Galerie überschreitet und es letztmalig verlässt. Die Handler müssen jedoch jedes Mal, wenn die Maus eines der Fotos innerhalb des umschließenden `<div>` berührt, ausgelöst werden. Mit den neuen Typen haben wir jedoch ein anderes Problem, da das `<div>` wiederholt ein- und ausgeblendet wird, bis wir eine zusätzliche Abfrage für die Eigenschaft `relatedTarget` des Objekts `event` einbauen. Selbst mit dieser zusätzlichen Abfrage werden schnelle Mausbewegungen auf und von einem Foto weg nicht zufriedenstellend behandelt, weil manchmal immer noch ein `<div>`-Element sichtbar bleibt, obwohl es ausgeblendet sein sollte.

### 10.2.1 Die jQuery-Delegationsmethoden verwenden

Ereignisdelegation kann frustrierend schwer zu handhaben sein, wenn die Aufgaben kompliziert werden. Glücklicherweise helfen uns die *Delegationsmethoden* von jQuery bei den harten Nüssen. Mit diesen Methoden erhält unser Code wieder die Leichtigkeit wie in Listing10–3:

```
$(document).ready(function() {  
    $('#gallery').delegate('div.photo', 'mouseenter mouseleave',  
        function(event) {  
            var $details = $(this).find('.details');  
            if (event.type == 'mouseenter') {  
                $details.fadeTo('fast', 0.7);  
            } else {  
                $details.fadeOut('fast');  
            }  
        });  
    });
```

**Listing 10–5**

Der Selektor `#gallery` bleibt wie in Listing10–4, die Ereignistypen jedoch sind wieder `mouseenter` und `mouseleave` wie in Listing10–3. Wenn wir `'div.photo'` an das erste Argument übergeben, weist die Methode `.delegate()` das Schlüsselwort `this` den zu diesem Selektor passenden Elementen innerhalb von `'#gallery'` zu.

### 10.2.2 Eine Delegationsmethode wählen

Als in jQuery 1.3 die Methode `.live()` eingeführt wurde, war sie als Ersatz für `.bind()` gedacht. Alle Entwickler, deren Ereignishandler mit später ins DOM integrierten Elementen arbeiten sollten, und diejenigen, die die Performance-Einbußen verhindern wollten, die das Binden eines Ereignishandlers an eine große Zahl von Elementen mit sich bringt, konnten einfach ihre `.bind()`-Methoden durch `.live()`-Methoden ersetzen und diese würden ebenfalls funktionieren.

Abgesehen vom Austausch von `.live()` durch `.bind()` ist der Code in Listing 10–6 identisch mit dem aus Listing 10–3:

```
$(document).ready(function() {
    $('div.photo')
        .live('mouseenter mouseleave', function(event) {
            var $details = $(this).find('.details');
            if (event.type == 'mouseenter') {
                $details.fadeTo('fast', 0.7);
            } else {
                $details.fadeOut('fast');
            }
        });
});
```

*Listing 10–6*

Trotz aller Annehmlichkeit ist `.live()` jedoch kein genaues Abbild von `.bind()` mit etwas zusätzlicher Delegationszauberei. Im Gegensatz zu `.bind()` kann `.live()` zum Beispiel nach den meisten DOM-Traversierungsmethoden nicht zuverlässig aufgerufen werden. Während also `($('div').filter('.photo')).bind('click', fn)` wie gedacht funktionieren würde, wäre das bei `($('div').filter('.photo')).live('click', fn)` nicht der Fall.

Hinzu kommt, dass die Methode `.live()` unter bestimmten Umständen zu den gleichen Performance-Einbußen führt wie ihr Gegenstück `.bind()`. Obwohl `.live()` das Binden von Handlern an viele Ereignisse verhindert, wählt es diese Elemente alle im Vorfeld aus. In unserem Beispiel der Fotogalerie ist das nicht von großer Bedeutung. Wenn wir Handler jedoch an bestimmte Elemente in einer Tabelle mit vielen Spalten und Hunderten von Zeilen binden, kann genau das Auswählen der Elemente dazu führen, dass das Skript nicht mehr reagiert. Zusätzlich stört es, dass `.live()` den Satz der durch die Funktion `$()` gesammelten passenden Elemente nicht einmal verwendet. Stattdessen benutzt es den Selektorausdruck und den aktuellen Stringwert, um ihn mit `event.target` oder seinen Nachfahren zu vergleichen.

Ein weiterer potenzieller Performance-Verlust von `.live()` besteht darin, dass es Ereignisse ans Dokument bindet. In einem DOM mit tief verschachtelten Elementen kann es zeitraubend sein, sich darauf zu verlassen, dass sich Ereignisse den

Weg durch viele Nachfahren bahnen. Mit `.delegate()` werden Ereignisse an Elemente gebunden, die dem Selektorausdruck in der `$()`-Funktion entsprechen, sodass der Teil der Seite, der das Ereignis feststellt, präziser sein kann und das Event Bubbling reduziert wird.

Aufgrund all dieser Performance-Bedenken – Auswählen der Elemente zu Beginn und exzessives Event Bubbling – haben sich manche Entwickler von `.live()` abgewandt und nutzen `.delegate()`, als es mit jQuery 1.4.2 verfügbar wurde. Trotzdem müssen wir nicht vollständig auf `.live()` verzichten. Wenn wir es vorsichtig einsetzen, entweder durch frühes Aufrufen oder indem wir einen Kontext dazu angeben, können wir die Annehmlichkeiten von `.live()` nutzen und seine Nachteile vermeiden.

### 10.2.3 Frühe Delegation

Ein Weg, Performance-Einbußen durch die Elementauswahl der `$()`-Funktion zu vermeiden, ist es, den `.live()`-Handler außerhalb von `$(document).ready()` zu platzieren. Wenn das Skript im `<head>` des Dokuments referenziert wird, wie in unserem Beispiel, hat der Selektor nur wenig zu tun, da zu diesem Zeitpunkt nur ein geringer Teil des DOM bereits registriert ist. Natürlich ist `document`, das `.live()` verwendet, immer verfügbar.

```
(function($) {
    $('div.photo').live('mouseenter mouseleave',
        function(event) {
            var $details = $(this).find('.details');
            if (event.type == 'mouseenter') {
                $details.fadeTo('fast', 0.7);
            } else {
                $details.fadeOut('fast');
            }
        });
})(jQuery);
```

**Listing 10–7**

Da wir nicht warten, bis das gesamte DOM bereit ist, können wir sicher sein, dass das Verhalten von `mouseenter` und `mouseleave` sich sofort auf alle `<div class="photo">`-Elemente auswirkt, sobald sie sich auf der Seite befinden. Um den Vorteil dieser Technik zu erkennen, stellen Sie sich einen `click`-Handler vor, der unter anderem die Standardaktion eines Links unterbindet. Wenn wir warten müssten, bis das Dokument bereit ist, würden wir Gefahr laufen, dass der Benutzer auf den Link klickt, bevor der Handler registriert ist und also die aktuelle Seite verlässt, statt dass er die erweiterte Behandlung durch das Skript erhält. In jedem Fall haben wir jetzt den Vorteil, das Ereignis frühzeitig zu binden, ohne eine komplexe DOM-Struktur durchgehen zu müssen.

### IIFE

Statt `$(document).ready()` verwenden wir einen *sofort aufgerufenen Funktionsaufruf* (Immediately Invoked Function Expression, IIFE) als Kapselung, wie in Kapitel 8 besprochen. Dies ermöglicht es uns, potenzielle Benennungskonflikte mit anderen Skripten zu vermeiden, wenn wir die Variablen und Funktionen darin definieren (da Variablen innerhalb der Funktionen gültig sind).

#### 10.2.4 Ein Kontextargument verwenden

Eine weitere Technik, die wir bei `.live()` anwenden können, besteht darin, der Funktion `$()` ein Kontextargument mitzugeben. Damit können wir das Event Bubbling genauso begrenzen wie mit `.delegate()`. Zum Beispiel können wir `'div.photo', $('#gallery')[0]'.live...` statt `'div.photo'.live...` schreiben. Dies erreichen wir durch Verwenden des Array-Index-Operators, wie in `'#gallery')[0]`, oder indem wir die zugrunde liegende DOM-Methode, wie `document.getElementById('gallery')`, selbst verwenden.

Sollte diese Genauigkeit notwendig sein, müssen wir das Skript aber zurück in einen `$(document).ready()`-Aufruf verlegen, solange es in `<head>` referenziert wird, wodurch wir den Vorteil der frühen Delegation verlieren. Aus diesem Grund verbleibt unser Skript in dem System, das in Listing 10–7 gezeigt wird.

### 10.3 Benutzerdefinierte Ereignisse

Die Ereignisse, die üblicherweise durch die DOM-Implementierungen der Browser ausgelöst werden, sind für jede interaktive Webanwendung von größter Wichtigkeit. Wir sind in unserem jQuery-Code aber nicht auf diesen Satz von Ereignissen festgelegt. Wir können unserem Repertoire unsere eigenen Ereignisse hinzufügen. Dies haben wir kurz in Kapitel 8, »*Plug-ins entwickeln*«, gestreift, als wir gesehen haben, wie jQuery-UI-Widgets Ereignisse auslösen. Hier werden wir nun erforschen, wie wir benutzerdefinierte bzw. eigene Ereignisse außerhalb von Plug-ins erzeugen und einsetzen.

Benutzerdefinierte Ereignisse müssen durch unseren Code manuell ausgelöst werden. Aus einer bestimmten Sicht sind sie wie selbst definierte reguläre Funktionen, in denen wir einen Codeblock ausführen können, wenn wir sie von anderer Stelle im Skript aufrufen. Der `.bind()`-Aufruf entspricht einer Funktionsdefinition und der `.trigger()`-Aufruf einem Funktionsaufruf.

Ereignishandler sind jedoch vom Code, der sie auslöst, entkoppelt. Das bedeutet, dass wir Ereignisse jederzeit auslösen können, ohne vorher zu wissen, was passiert. Wir können vielleicht einen einfach gebundenen Ereignishandler auslösen wie eine normale Funktion. Wir können jedoch auch viele Handler auslösen oder gar keinen.

Um dies zu illustrieren, können wir unsere Ajax-Ladefunktion überarbeiten, sodass sie ein benutzerdefiniertes Ereignis verwendet. Wir lösen immer dann ein nextPage-Ereignis aus, wenn der Benutzer weitere Fotos anfordert, und binden neue Handler, die dieses Ereignis beobachten und die Aufgaben durchführen, die früher vom .click-Handler erledigt wurden:

```
$(document).ready(function() {
    $('#more-photos').click(function() {
        $(this).trigger('nextPage');
        return false;
    });
});
```

**Listing 10–8**

Der .click()-Handler erledigt jetzt selbst nur noch wenig Arbeit. Nach Auslösen des eigenen Ereignisses verhindert er das Standardverhalten, indem er false zurückgibt. Die schwere Arbeit wird den neuen Ereignishandlern für das nextPage-Ereignis wie folgt übertragen:

```
(function($) {
    $(document).bind('nextPage', function() {
        var url = $('#more-photos').attr('href');
        if (url) {
            $.get(url, function(data) {
                $('#gallery').append(data);
            });
        }
    });

    var pageNum = 1;
    $(document).bind('nextPage', function() {
        pageNum++;
        if (pageNum < 20) {
            $('#more-photos')
                .attr('href', 'pages/' + pageNum + '.html');
        }
        else {
            $('#more-photos').remove();
        }
    });
})(jQuery);
```

**Listing 10–9**

Ein Großteil des Codes stimmt mit dem in Listing10–1 überein. Der größte Unterschied besteht in der Aufteilung einer einzelnen Funktion auf zwei. Dies dient einfach der Veranschaulichung, dass ein einzelnes Ereignis mehrere verknüpfte Handler auslösen kann.

Das Beispiel zeigt auch eine weitere Anwendung des Event Bubbling. Die nextPage-Handler können an den Link gebunden werden, der das Ereignis auslöst, aber wir müssten damit warten, bis das DOM vollständig ist. Stattdessen binden wir die Handler an das Dokument selbst, was sofort möglich ist, sodass wir das Binden außerhalb von `$(document).ready()` erledigen können. Es handelt sich hier um dasselbe Prinzip, das wir in Listing 10–6 genutzt haben, als wir die Methode `.live()` außerhalb von `$(document).ready()` verschoben haben. Das Ereignis bewegt sich nach oben, und solange kein anderer Handler die Ereignisweitergabe beendet, werden die Handler ausgelöst.

### 10.3.1 Unendliches Scrollen

So wie mehrere Ereignishandler auf dasselbe ausgelöste Ereignis reagieren können, kann dasselbe Ereignis auf verschiedene Weise ausgelöst werden. Wir zeigen dies durch eine unendliche Scrollfunktion auf unserer Webseite. Diese beliebte Technik übergibt das Laden von Inhalten an den Rollbalken des Benutzers und holt immer dann neuen Inhalt, wenn der Benutzer an das Ende des bereits geladenen Inhalts kommt.

Wir beginnen mit einer einfachen Implementierung und verbessern sie dann schrittweise. Die Grundidee besteht darin, das scroll-Ereignis zu beobachten, die aktuelle Position des Rollbalkens während des Scrollens zu messen und den Inhalt bei Bedarf nachzuladen:

```
(function($) {
    var $window = $(window);
    function checkScrollPosition() {
        var distance = $window.scrollTop() + $window.height();
        if ($('#container').height() <= distance) {
            $(document).trigger('nextPage');
        }
    }
    $(document).ready(function() {
        $window.scroll(checkScrollPosition).scroll();
    });
})(jQuery);
```

#### *Listing 10–10*

Die neue Funktion `checkScrollPosition()` wird als Handler für das scroll-Ereignis des Fensters verwendet. Sie berechnet den Abstand vom Dokumentanfang zum Fensterende und vergleicht die Distanz mit der Gesamthöhe des Hauptcontainers im Dokument. Sind beide gleich, müssen wir die Seiten mit zusätzlichen Fotos füllen, also lösen wir das Ereignis `nextPage` aus.

Sobald wir den scroll-Handler gebunden haben, lösen wir ihn mit einem Aufruf von `.scroll()` aus. Dadurch wird der Vorgang gestartet, sodass umgehend eine Ajax-Anfrage gesendet wird, wenn sich die Seite nicht gleich mit Fotos füllt.

### 10.3.2 Benutzerdefinierte Ereignisparameter

Wenn wir Funktionen definieren, können wir eine beliebige Anzahl von Parametern setzen, die beim Aufruf der Funktion mit Argumentwerten gefüllt werden. Wenn wir ein eigenes Ereignis auslösen, könnten wir ebenfalls zusätzliche Informationen an die Ereignishandler weitergeben wollen. Dies erreichen wir über benutzerdefinierte Ereignisparameter.

Der erste für den Ereignishandler definierte Parameter ist, wie wir gesehen haben, das DOM-Ereignisobjekt, das durch jQuery verbessert und erweitert wurde. Alle zusätzlich von uns definierten Parameter dienen unserem eigenen Gebrauch.

Um diesen Vorgang zu betrachten, fügen wir dem nextPage-Ereignis eine neue Option hinzu, mit der wir die Seite hinunterscrollen, um den neu hinzugefügten Inhalt zu sehen:

```
(function($) {
    $(document).bind('nextPage',
        function(event, scrollTopVisible) {
            var url = $('#more-photos').attr('href');
            if (url) {
                $.get(url, function(data) {
                    var $data = $(data).appendTo('#gallery');
                    if (scrollTopVisible) {
                        var newTop = $data.offset().top;
                        $(window).scrollTop(newTop);
                    }
                    checkScrollPosition();
                });
            }
        });
});
```

**Listing 10–11**

Jetzt haben wir dem Ereignis-Callback einen `scrollTopVisible`-Parameter hinzugefügt. Der Wert dieses Parameters legt fest, ob wir die neue Funktionalität nutzen, die die Position des neuen Inhalts misst und dahin scrollt. Die Messung erfolgt einfach mit der `.offset()`-Methode, die die oberen linken Koordinaten des neuen Inhalts zurückgibt. Um uns auf der Seite nach unten zu bewegen, rufen wir die Methode `.scrollTop()` auf.

Jetzt müssen wir dem neuen Parameter ein Argument übergeben. Dazu müssen wir nur einen zusätzlichen Wert angeben, wenn wir das Ereignis mit `.trigger()` einrichten. Ist `newPage` durch das Scrollen ausgelöst worden, soll das neue Verhalten nicht auftreten, da der Benutzer den Rollbalken manuell bewegt. Wenn andererseits der Link `MORE PHOTOS` angeklickt wird, wollen wir, dass die neu hinzugefügten Fotos auf dem Bildschirm angezeigt werden, sodass wir dem Handler den Wert `true` wie folgt übergeben können:

```
$(document).ready(function() {
    $('#more-photos').click(function() {
        $(this).trigger('nextPage', [true]);
        return false;
    });

    $window.scroll(checkScrollPosition).scroll();
});
```

**Listing 10–12**

Im Aufruf von `.trigger()` geben wir nun ein Array mit Werten an, die den Ereignishandlern übergeben werden sollen. In diesem Fall wird dem Parameter `scrollToVisible` des Ereignishandlers in Listing 10–11 der Wert `true` übergeben.

Beachten Sie, dass eigene Ereignisparameter auf beiden Seiten der Transaktion optional sind. Wir haben zwei `.trigger()`-Aufrufe in unserem Code, von denen nur einer Argumentwerte übergibt. Wird der andere aufgerufen, führt das nicht zu einem Fehler, aber den Parametern wird der Wert null zugewiesen. So ist auch das Fehlen des `scrollToVisible`-Parameters in unseren `.bind('nextPage')`-Aufrufen kein Fehler. Existiert ein Parameter nicht, wenn ein Argument übergeben wird, wird das Argument einfach ignoriert.

## 10.4 Ereignisse drosseln

Ein größeres Problem der unendlichen Scrollfunktion aus Listing 10–10 ist ihre Auswirkung auf die Performance. Auch wenn der Code kurz ist, muss die Funktion `checkScrollPosition()` einige Aufgaben erledigen, um die Abmessungen der Seite und des Fensters zu ermitteln. Dieser Aufwand kann sich schnell kumulieren, da in einigen Browsern das `scroll`-Ereignis während des Scrollens wiederholt ausgelöst wird. Das Resultat dieser Kombination kann ein zerhackter oder langsamer Aufbau sein.

Verschiedene native Ereignisse haben das Potenzial für häufiges Auslösen. Übliche Fallstricke sind `scroll`, `resize` und `mousemove`. Um dies zu berücksichtigen, müssen wir unsere zeitaufwendigen Berechnungen einschränken, sodass sie nur nach manchen der Ereignisinstanzen erfolgen statt jedes Mal. Diese Technik ist als *Ereignisdrosselung* bzw. *Event Throttling* bekannt.

```
$(document).ready(function() {
    var timer = 0;
    $window.scroll(function() {
        if (!timer) {
            timer = setTimeout(function() {
                checkScrollPosition();
                timer = 0;
            }, 250);
        }
    }).scroll();
});
```

**Listing 10–13**

Statt `checkScrollPosition()` direkt als Ereignishandler für `scroll` einzusetzen, verwenden wir die JavaScript-Funktion `setTimeout`, um den Aufruf um 250 Millisekunden zu verzögern. Noch wichtiger: Wir prüfen zuerst, ob bereits ein Timer läuft, bevor wir eine Aktion ausführen. Da das Prüfen einer einfachen Variablen extrem schnell erfolgt, kehren die meisten Aufrufe des Ereignishandlers nahezu sofort zurück. Der Aufruf `checkScrollPosition()` findet nur statt, wenn ein Timer abgelaufen ist, was höchstens alle 250 Millisekunden der Fall ist.

Wir können den Wert von `setTimeout()` so einstellen, dass er einen sinnvollen Kompromiss zwischen schnellem Feedback und geringer Auswirkung auf die Performance bietet. Jetzt ist unser Skript webkonform.

#### 10.4.1 Andere Arten der Drosselung

Die implementierte Drosselungstechnik ist effizient und einfach, aber nicht die einzige Lösung. Abhängig von der Performance-Charakteristik der gedrosselten Aktion und der typischen Interaktion mit der Seite wollen wir vielleicht einen einzelnen Timer für die Seite verwenden, statt einen für jedes Ereignis zu erstellen:

```
$(document).ready(function() {
    var scrolled = false;
    $window.scroll(function() {
        scrolled = true;
    });
    setInterval(function() {
        if (scrolled) {
            checkScrollPosition();
            scrolled = false;
        }
    }, 250);
    checkScrollPosition();
});
```

**Listing 10–14**

Anders als unser Drosselungscode verwendet diese Polling-Lösung einen einzelnen `setInterval()`-Aufruf, um den Status der `scrolled`-Variable alle 250 Millisekunden zu prüfen. Immer, wenn ein Scroll-Ereignis auftritt, wird `scrolled` auf `true` gesetzt, was sicherstellt, dass beim nächsten Durchgang des Intervalls `checkScrollPosition()` aufgerufen wird. Das Ergebnis gleicht dem von Listing 10–13.

Eine dritte Lösung, um die aufgewendete Rechenzeit für häufige Ereignisse zu beschränken, ist das *Entprellen*. Diese Technik, benannt nach der Nachbehandlung, die wiederholt auftretende Signale von elektrischen Schaltern benötigen, ermöglicht, dass nur auf ein einzelnes, letztes Signal reagiert wird, auch wenn mehrere aufgetreten sind. Ein Beispiel dazu sehen wir in Kapitel 13, »Ajax für Fortgeschrittene«.

## 10.5 Spezielle Ereignisse

Einige Ereignisse, wie `mouseenter` und `ready`, sind jQuery-intern als spezielle Ereignisse ausgelegt. Solche Ereignisse können zu verschiedenen Zeitpunkten im Lebenszyklus eines Ereignishandlers in Aktion treten. Sie können gebunden oder entbunden auf Handler reagieren, und sie können sogar unterdrückbare Standardverhalten aufweisen wie solche bei Links oder bei zu übertragenden Formularen. Die API für spezielle Ereignisse ermöglicht es uns, ausgefeilte neue Ereignisse zu erstellen, die sich sehr ähnlich den nativen DOM-Ereignissen verhalten.

Das Drosselungsverhalten, das wir für das Scrollen in Listing 10–13 implementiert haben, ist nützlich und könnte in standardisierter Form auch anderswo eingesetzt werden. Dies lässt sich erreichen, indem wir ein neues Ereignis erstellen, das die Drosselungstechnik mit speziellen Ereignis-Hooks kapselt.

Um ein spezielles Verhalten für ein Ereignis zu implementieren, fügen wir dem Objekt `$.event.special` eine Eigenschaft hinzu. Diese Eigenschaft, deren Schlüssel unser Ereignisname ist, hat einen Wert, der selbst ein Objekt ist. Dieses spezielle Ereignisobjekt verfügt über fünf besondere Eigenschaften, die wir wenn gewünscht definieren können, und die jeweils eine Callback-Funktion enthalten:

1. `add` wird jedes Mal aufgerufen, wenn ein Handler für dieses Ereignis gebunden wird.
2. `remove` wird jedes Mal aufgerufen, wenn eine Handler für das Ereignis entbunden wird.
3. `setup` wird aufgerufen, wenn ein Handler für das Ereignis gebunden wird, aber nur, wenn keine anderen Handler für das Ereignis an das Element gebunden sind.
4. `teardown` ist das Gegenteil von `setup` und wird aufgerufen, wenn der letzte Handler des Ereignisses entbunden wird.
5. `_default` ist das Standardverhalten des Ereignisses und wird immer aufgerufen, wenn kein Handler die Standardaktion verhindert.

Diese Callbacks können sehr vielfältig eingesetzt werden. In unserem Beispielcode untersuchen wir das weitverbreitete Szenario, ein Ereignis automatisch aufgrund eines Browserzustands auszulösen. Es wäre Verschwendug, den Status zu überwachen und Ereignisse auszulösen, wenn dafür keine Handler bereitstünden. Daher setzen wir den setup-Callback dazu ein, die Vorgänge nur zu starten, wenn sie benötigt werden:

```
(function($) {
    $.event.special.throttledScroll = {
        setup: function(data) {
            var timer = 0;
            $(this).bind('scroll.throttledScroll', function(event) {
                if (!timer) {
                    timer = setTimeout(function() {
                        $(this).triggerHandler('throttledScroll');
                        timer = 0;
                    }, 250);
                }
            });
        },
        teardown: function() {
            $(this).unbind('scroll.throttledScroll');
        }
    };
})($);
```

**Listing 10–15**

Für unser Scroll-Drosselungs-Ereignis müssen wir einen regulären scroll-Handler binden, der die setTimeout-Technik aus Listing 10–13 verwendet. Immer, wenn der Timer abgelaufen ist, wird das benutzerdefinierte Ereignis ausgelöst. Da wir je Element nur einen Timer benötigen, erfüllt der setup-Callback unsere Anforderungen. Indem wir den scroll-Handler mit einem Namensraum versehen, können wir den Handler einfach entfernen, wenn teardown aufgerufen wird.

Um dieses neue Verhalten zu verwenden, müssen wir die Handler nur an das Ereignis throttledScroll binden:

```
$(document).ready(function() {
    $window
        .bind('throttledScroll', checkScrollPosition)
        .trigger('throttledScroll');
});
```

**Listing 10–16**

Das vereinfacht den Code zum Binden von Ereignissen erheblich und gibt uns einen hübschen, wiederverwendbaren Beschleunigungsmechanismus.

### 10.5.1 Weitere Informationen zu speziellen Ereignissen

Auch wenn dieses Kapitel die erweiterten Techniken zum Umgang mit Ereignissen behandelt und die Erstellung spezieller Ereignisse ein äußerst hilfreiches Werkzeug darstellt, so sprengt aber eine detaillierte Vorstellung den Rahmen dieses Buches. Das vorgestellte Beispiel `throttledScroll` zeigt die einfachste und verbreitetste Verwendung dieser Funktion. Andere denkbare Anwendungen könnten sein:

- Das Ereignisobjekt zu modifizieren, sodass Ereignishandler darüber verschiedene Informationen bekommen können.
- Ereignisse zu erstellen, die im DOM auftreten, um das Verhalten bestimmter, zugeordneter Elemente auszulösen.
- Auf neue, browserspezifische Ereignisse zu reagieren, die keine Standard-DOM-Ereignisse sind, und mit jQuery-Code darauf zu reagieren, als wenn sie es wären.
- Die Art und Weise zu verändern, wie Event Bubbling und Ereignisdelegation verwendet werden.

Viele dieser Aufgaben können ziemlich kompliziert werden. Einen tieferen Einblick in die Möglichkeiten von speziellen Ereignissen finden Sie in Ben Almans Artikel »jQuery Special Events« unter folgender URL: <http://benalman.com/news/2010/03/jquery-special-events/>.

## 10.6 Zusammenfassung

Das jQuery-Ereignissystem kann sehr leistungsfähig sein, wenn wir es voll aus schöpfen. In diesem Kapitel haben wir die verschiedenen Aspekte des Systems betrachtet, wie Methoden zur Ereignisdelegation, benutzerdefinierte Ereignisse und das Framework für spezielle Ereignisse. Wir haben außerdem Wege gefunden, Klippen bei der Delegation und bei häufig ausgelösten Ereignissen zu umschiffen.

### 10.6.1 Literatur

Eine vollständige Liste der Ereignismethoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen Dokumentation unter folgender URL: <http://api.jquery.com/>.

## 10.7 Übungsaufgaben

Um die folgenden Übungsaufgaben durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Wenn der Benutzer auf ein Foto klickt, entfernen Sie die Klasse `selected` vom `<div>`-Element des Fotos. Achten Sie darauf, dass dieses Verhalten auch bei Fotos funktioniert, die später über den Link NEXT PAGE hinzugekommen sind.
2. Fügen Sie ein neues, benutzerdefiniertes Ereignis namens `pageLoaded` hinzu, das ausgelöst wird, wenn der Seite ein neuer Satz Fotos hinzugefügt wurde.
3. Stellen Sie unten am Bildschirm eine *Loading*-Meldung mit den Handlern `nextPage` und `pageLoaded` dar, die angezeigt wird, während die neue Seite geladen wird.
4. Binden Sie einen `mousemove`-Handler an die Fotos, der die aktuelle Mausposition speichert (mittels `console.log()`).
5. Verändern Sie den Handler so, dass er nicht häufiger als fünf Mal je Sekunde Daten speichert.
6. *Schwierig:* Erstellen Sie ein neues, spezielles Ereignis namens `triplecheck`, das ausgelöst wird, wenn die Maustaste innerhalb von 500 Millisekunden drei Mal gedrückt wird. Um das Ereignis zu testen, binden Sie einen `triplecheck`-Handler an das `<h1>`-Element, der den Inhalt von `<div id="gallery">` anzeigt oder verbirgt.

# 11 Anspruchsvolle Effekte

Seit wir in Kapitel 4, »*Formatierung und Animation*«, einiges über die jQuery-Funktionen zur Animation erfahren haben, sind wir auf viele Anwendungsfälle gestoßen. Wir können Objekte auf der Seite mit Leichtigkeit verbergen und wieder anzeigen, wir können Elemente in der Größe verändern und wir können sie neu positionieren. Die Effektbibliothek ist aber wesentlich vielseitiger und enthält weitaus mehr Techniken und Sonderfunktionen, als wir bislang gesehen haben.

In diesem Kapitel untersuchen wir einige dieser erweiterten Fähigkeiten. Wir lernen, wie wir laufende Animationen sowohl durch Reaktionen auf Fortschritt und Abschluss der Animationen als auch durch Abfragen ihres aktuellen Animationsstatus besser im Auge behalten. Wir werden sehen, wie wir laufende Animationen unterbrechen und wie wir alle Effekte auf unserer Seite global verändern. Wir werden auch tiefer ins Thema Easing einsteigen, wodurch wir den Ablauf der Effekte noch feiner steuern können.

## 11.1 Animation – Teil 2

Um unsere Erinnerungen an die jQuery-Effektmethoden aufzufrischen und eine Grundlage für die Themen dieses Kapitels zu schaffen, beginnen wir mit einer einfachen Hover-Animation.

Wir verwenden ein Dokument mit einer Anzahl von Fotominiaturen, die sich jeweils etwas vergrößern, wenn die Maus darüberrollt, und die wieder kleiner werden, wenn die Maus sie verlässt. Der folgende HTML-Code enthält auch einige Textinformationen, die momentan noch verborgen sind, die wir aber später in diesem Kapitel verwenden werden:

```
<div class="team">
  <div class="member">
    
    <div class="name">Rey Bango</div>
    <div class="location">Florida</div>
    <p class="bio">Rey Bango is a consultant living in South
      Florida, specializing in web application development...</p>
  </div>
```

```
<div class="member">
  
  <div class="name">Scott González</div>
  <div class="location">North Carolina</div>
  <div class="position">jQuery UI Development Lead</div>
  <p class="bio">Scott is a web developer living in
    Raleigh, NC...</p>
</div>
<!-- Code geht hier weiter ... -->
</div>
```

HTML und CSS aus diesem Beispiel ergeben eine senkrecht angeordnete Reihe von Bildern, wie im folgenden Screenshot gezeigt. Der mit den Bildern verbundene Text wird momentan durch das CSS unterdrückt.

### Executive Board

The Executive Board is responsible for the day-to-day operations of the jQuery project, and has powers delegated to it by our governance plan or a regular vote of the voting membership. The Executive Board is made up of seven members of the voting membership, elected twice annually by the voting membership, in October and April.



Um die Bildgröße zu verändern, erhöhen wir die Höhe und Breite von 75 auf 85 Pixel. Gleichzeitig reduzieren wir seinen Innenrand von 5 auf 0 Pixel, damit das Bild immer zentriert bleibt:

```
$(document).ready(function() {
  $('div.member')
    .bind('mouseenter mouseleave', function(event) {
      var size = event.type == 'mouseenter' ? 85 : 75;
      var padding = event.type == 'mouseenter' ? 0 : 5;
      $(this).find('img').animate({
        width: size,
        height: size,
```

```
paddingTop: padding,  
paddingLeft: padding  
});  
})  
});
```

**Listing 11-1**

Hier wiederholen wir ein Muster aus Kapitel 10. Da eine Menge der von uns ausgeführten Arbeit beim Bewegen der Maus in die Region dieselbe ist wie beim Verlassen, kombinieren wir die Handler für `mouseenter` und `mouseleave` in einer Funktion, statt `.hover()` mit zwei separaten Callbacks aufzurufen. In diesem zusammengesetzten Handler legen wir die Werte für `size` und `padding` abhängig davon fest, welches Ereignis ausgelöst wurde, und geben die Eigenschaftswerte an die `.animate()`-Methode weiter.

Wenn sich jetzt der Mauszeiger über dem Bild befindet, ist es ein wenig größer als der Rest, wie der folgende Screenshot zeigt:



## 11.2 Animationen beobachten und unterbrechen

Schon unsere einfache Animation deckt ein Problem auf. Solange der Benutzer die Maus vorsichtig über und von den Bildern wegzieht, um wiederholte `mouseenter`- und `mouseleave`-Ereignisse nicht zu schnell auszulösen, funktioniert die Animation wie gewünscht. Wenn die Ereignisse jedoch schnell und häufiger kommen, sehen wir, dass die Bilder auch nach dem letzten Ereignis noch mehrfach größer und kleiner werden. Dies geschieht, da – wie in Kapitel 4 besprochen – Animationen eines Elements in einer Warteschleife landen und nacheinander abgearbeitet werden. Die erste Animation wird sofort gestartet, innerhalb einer zugewiesenen Dauer abgearbeitet und dann aus der Warteschlange entfernt. Jetzt ist die folgende Animation vorn in der Reihe, wird aufgerufen, abgearbeitet, entfernt usw., bis die Warteschlange leer ist.

Es gibt viele Fälle, in denen diese Animationswarteschlange, jQuery-intern `fx` genannt, erwünscht ist. Bei Hover-Animationen, wie in unserem Fall, muss sie jedoch umgangen werden.

### 11.2.1 Den Animationsstatus bestimmen

Eine Möglichkeit, die ungewünschte Animationswarteschlange zu umgehen, besteht darin, den speziellen jQuery-Selektor `:animated` einzusetzen. Innerhalb des `mouseenter-/mouseleave-Ereignishandlers` können wir den Selektor verwenden, um das Bild zu prüfen und festzustellen, ob es zurzeit animiert wird. Wenn die Maus des Benutzers das `<div>`-Element erreicht, wird das Bild nur dann animiert, wenn das noch nicht der Fall ist. Wenn die Maus verschwindet, erscheint die Animation unabhängig vom Status, da wir möchten, dass die Originalgröße und der Innenrand wie folgt wiederhergestellt werden:

```
$(document).ready(function() {
    $('#div.member')
        .bind('mouseenter mouseleave', function(event) {
            var $image = $(this).find('img');
            if (!$image.is(':animated'))
                || event.type == 'mouseleave') {
                var size = event.type == 'mouseenter' ? 85 : 75;
                var padding = event.type == 'mouseenter' ? 0 : 5;
                $image.animate({
                    width: size,
                    height: size,
                    paddingTop: padding,
                    paddingLeft: padding
                });
            }
        });
});
```

#### **Listing 11-2**

Wir haben die weiterlaufenden Animationen wie in Listing 11-1 damit erfolgreich vermieden, doch es gibt noch immer Verbesserungspotenzial. Wenn die Maus das `<div>`-Element schnell erreicht und wieder verlässt, muss das Bild trotzdem noch eine vollständige `mouseenter`-Animation durchlaufen (also größer werden), bevor die `mouseleave`-Animation (Bild verkleinern) beginnt. Das ist sicher nicht ideal, aber durch die `:animated`-Prüfung sind wir in ein noch größeres Problem geraten: Wenn die Maus das `<div>`-Element erreicht, während das Bild verkleinert wird, schlägt das anschließende Vergrößern fehl. Nur ein nach dem Stopp der Animation auftretendes `mouseleave` oder `mouseenter` führt zu einer neuen Animation. Während eine Statusprüfung eines Elements mit dem `:animated`-Selektor in manchen Situationen sinnvoll ist, hilft sie uns in diesem Fall nicht viel.

### 11.2.2 Eine laufende Animation anhalten

Glücklicherweise besitzt jQuery eine Methode, uns bei beiden Problemen aus Listing 11–2 zu unterstützen. Die Methode `.stop()` kann eine Animation in der Spur halten. Um sie zu nutzen, kehren wir zum Code aus Listing 11–1 zurück und fügen einfach `.stop()` zwischen `.find()` und `.animate()` ein:

```
$(document).ready(function() {
    $('div.member').bind('mouseenter mouseleave', function(event) {
        var size = event.type == 'mouseenter' ? 85 : 75;
        var padding = event.type == 'mouseenter' ? 0 : 5;
        $(this).find('img').stop().animate({
            width: size,
            height: size,
            paddingTop: padding,
            paddingLeft: padding
        });
    });
});
```

**Listing 11–3**

Es ist durchaus erwähnenswert, dass wir die Animation anhalten, *bevor* wir mit der neuen beginnen. Wenn die Maus sich jetzt wiederholt in den Bereich bewegt und ihn wieder verlässt, ist der unerwünschte Effekt unserer früheren Versuche verschwunden. Die aktuelle Animation wird immer sofort abgeschlossen, sodass es nie mehr als eine in der fx-Warteschlange gibt. Steht die Maus dann still, wird die letzte Animation abgeschlossen und das Bild erscheint entweder in voller Größe (`mouseenter`) oder in den Ursprungsabmessungen (`mouseleave`), abhängig vom zuletzt ausgelösten Ereignis.

#### Vorsicht beim Anhalten von Animationen

Da die `.stop()`-Methode per Voreinstellung Animationen an ihren aktuellen Positionen einfriert, kann das zu überraschenden Ergebnissen führen, wenn schnelle Animationsmethoden ins Spiel kommen. Vor der Animation berechnen diese Methoden den Endwert und animieren bis dorthin. Wenn beispielweise `.slideDown()` mit `.stop()` mitten in der Animation angehalten, und dann `.slideUp()` aufgerufen wird, führt der nächste Aufruf von `.slideDown()` dazu, dass das Element nur bis zu der Position bewegt wird, an der es das letzte Mal angehalten wurde. Um dieses Problems zu verhindern, nimmt die Methode `.stop()` zwei boolesche (`true/false`) Argumente an, von denen das zweite als `goToEnd` bekannt ist. Wenn wir dieses Argument auf `true` setzen, wird die aktuelle Animation nicht nur angehalten, sondern springt sofort zum Endwert. Das kann natürlich immer noch komisch aussehen. Eine bessere Lösung wäre es, den Endwert in einer Variablen zu speichern und explizit bis dort mittels `.animate()` zu animieren, statt jQuery die Festlegung dieses Werts zu überlassen.

## 11.3 Globale Effekteigenschaften

Das Effektmodul in jQuery enthält ein praktisches `$.fx`-Objekt, auf das wir zugreifen können, wenn wir die Charakteristik unserer Animationen durchgängig ändern möchten. Obwohl einige der Objekteigenschaften undokumentiert und nur zum Gebrauch innerhalb der Bibliothek bestimmt sind, stehen uns andere als Werkzeug für globale Änderungen an unseren Animationen zur Verfügung.

### 11.3.1 Globales Deaktivieren aller Effekte

Wir haben bereits besprochen, wie laufende Animationen angehalten werden können, aber was tun, wenn wir alle Animationen deaktivieren wollen? Wir könnten Animationen beispielsweise per Voreinstellung aktiviert haben, sie jedoch bei weniger leistungsfähigen Geräten, wie Smartphones, abschalten wollen, da sie die Benutzer stören oder einfach schlecht aussehen. Für diesen Zweck können wir die Eigenschaft `$.fx.off` auf `true` setzen. Zu Demonstrationszwecken blenden wir eine vorher verborgene Schaltfläche ein, um dem Benutzer das Ein- und Ausschalten von Animationen zu ermöglichen:

```
$('#fx-toggle').show().bind('click', function() {  
    $.fx.off = !$._fx.off;  
});
```

**Listing 11-4**

Die versteckte Schaltfläche wird zwischen dem einleitenden Absatz und den nachfolgenden Bildern angezeigt:

#### Executive Board

The Executive Board is responsible for the day-to-day operations of the jQuery project, and has powers delegated to it by our governance plan or a regular vote of the voting membership. The Executive Board is made up of seven members of the voting membership, elected twice annually by the voting membership, in October and April.

[Toggle Animations](#)



Wenn der Benutzer auf diese Schaltfläche klickt, um die Animationen wie beispielsweise unsere sich vergrößernden und verkleinernden Bilder auszuschalten, erscheinen alle nachfolgenden Animationen mit einer Dauer von 0 und alle Callback-Funktionen werden quasi sofort ausgeführt.

### 11.3.2 Feineinstellung der Animationsübergänge

Das `$.fx`-Objekt verfügt auch über eine `interval`-Eigenschaft, die die Anzahl der Millisekunden zwischen den Animationsschritten bestimmt. Der Vorgabewert für `$.fx.interval` ist 13, was ungefähr einer Framerate von 77 Bildern pro Sekunde entspricht. Da nicht alle JavaScript-Timer so präzise sind und zwischenzeitlich unterbrochen werden können, ist die tatsächliche Framerate nicht konstant. Manche Entwickler haben berichtet, dass eine Erhöhung des Intervalls (oder eine Verringerung der Framerate) die Animationsübergänge weicher erscheinen lässt, besonders auf weniger leistungsfähigen Geräten, da es die Anzahl der Funktionen reduziert, die während einer Animation aufgerufen werden. Sichtbare Veränderungen lassen sich jedoch nur schwer bestätigen, weshalb Änderungen am Intervall die Mühe nicht wert sind. Zudem verwenden einige jQuery-Versionen eine »timerlose« DOM-Methode namens `requestAnimationFrame()` in Browsern, die sie unterstützen, statt `setInterval()` einzusetzen. Die Änderung von `$.fx.interval` hat also in diesen Fällen keine Wirkung.

### 11.3.3 Die Effektdauer festlegen

Eine weitere Eigenschaft des `$.fx`-Objekts ist `speeds`. Diese Eigenschaft ist selbst ein Objekt, das aus drei Eigenschaften besteht, wie dieser Ausschnitt der jQuery-Kerndatei zeigt:

```
speeds: {  
    slow: 600,  
    fast: 200,  
    // Standardgeschwindigkeit  
    _default: 400  
}
```

Wie haben bereits gelernt, dass alle Animationsmethoden von jQuery ein optionales Argument für *Geschwindigkeit* bzw. *Dauer* besitzen. Im `$.fx.speeds`-Objekt erkennen wir, dass die Strings `slow` und `fast` mit 600 und 200 Millisekunden belegt sind. Jedes Mal, wenn eine Animationsmethode aufgerufen wird, durchläuft jQuery die folgenden Schritte in dieser Reihenfolge:

1. Prüfen, ob `$.fx.off` auf `true` steht. Wenn ja, *Dauer* auf 0 setzen.
2. Prüfen, ob die *Dauer* als Zahl übergeben wird. Wenn ja, *Dauer* auf die Anzahl der Millisekunden setzen.
3. Prüfen, ob der übergebene Wert für *Dauer* einem der Eigenschaftenschlüssel des `$.fx.speeds`-Objekts entspricht. Wenn ja, *Dauer* auf den Wert dieser Eigenschaft setzen.
4. Wenn *Dauer* nicht über eine der vorherigen Prüfungen gesetzt werden kann, erhält das Argument den Wert von `$.fx.speeds._default`.

Mit diesen Informationen wissen wir, dass die Übergabe jedes Strings mit Ausnahme von `slow` und `fast` zu einer *Dauer* von 400 Millisekunden führt. Wir sehen auch, dass das Hinzufügen unserer eigenen Geschwindigkeit genauso leicht ist, wie `$.fx.speeds` eine andere Eigenschaft zuzuweisen. Wenn wir z.B. `$.fx.speeds.crawl = 1200` schreiben, können wir `crawl` für alle speed-Argumente von Animationsmethoden verwenden, die 1200 Millisekunden laufen sollen:

```
$(someElement).animate({width: '300px'}, 'crawl');
```

Auch wenn es nicht leichter ist, `crawl` zu schreiben als 1200, können eigene Geschwindigkeiten in größeren Projekten sehr praktisch werden, wenn viele Animationen die gleiche Geschwindigkeit haben sollen. In diesen Fällen müssen wir nur den Wert von `$.fx.speeds.crawl` ändern, statt im gesamten Projekt die 1200 zu suchen und an den entsprechenden Stellen mit Animationen durch einen anderen Wert zu ersetzen.

Obwohl eigene Geschwindigkeiten nützlich sein können, ist es vermutlich noch besser, die Vorgabegeschwindigkeit verändern zu können. Wir erreichen das durch Setzen der `_default`-Eigenschaft:

```
$.fx.speeds._default = 250;
```

#### ***Listing 11–5***

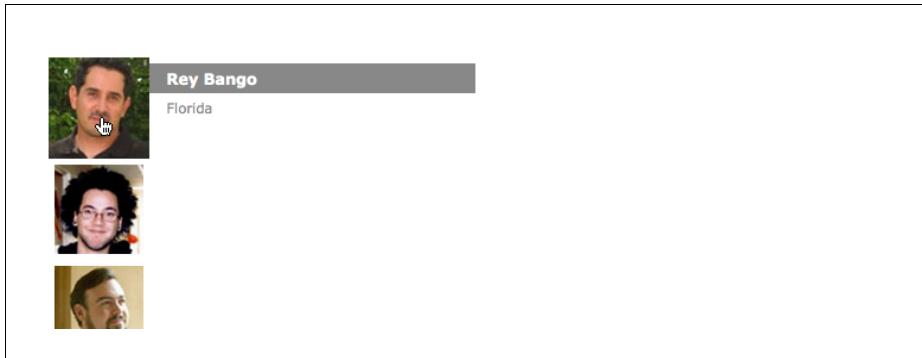
Damit haben wir eine neue, schnellere Vorgabegeschwindigkeit definiert und neu hinzugefügte Animationen werden diese verwenden, solange wir ihre Dauer nicht verändern. Um das in Aktion zu sehen, bauen wir ein weiteres, interaktives Element auf der Seite ein. Wenn der Benutzer auf eines der Portraits klickt, sollen Informationen zu dieser Person eingeblendet werden. Wir schaffen die Illusion, dass die Informationen sich aus dem Portrait heraus entfalten, indem wir sie von der Unterseite des Portraits in ihre endgültige Position bewegen:

```
$(document).ready(function() {
    function showDetails() {
        $(this).find('div').css({
            display: 'block',
            left: '-300px',
            top: 0
        }).each(function(index) {
            $(this).animate({
                left: 0,
                top: 25 * index
            });
        });
    }

    $('div.member').click(showDetails);
});
```

#### ***Listing 11–6***

Wenn eine Person auf unserer Liste angeklickt wird, verwenden wir die Funktion `showDetails()` als Handler. Diese Funktion setzt zuerst die `<div>`-Elemente auf ihre Startposition unterhalb des jeweiligen Portraits. Dann animiert sie jedes Element in seine endgültige Position. Durch Aufrufen von `.each()` können wir die jeweilige `top`-Position des Elements berechnen:



Da die `.animate()`-Aufrufe von verschiedenen Elementen kommen, geschehen sie gleichzeitig, statt in der Warteschlange zu landen. Außerdem werden sie alle mit der neuen Vorgabe von 250 Millisekunden ausgeführt, da die Aufrufe keine Dauer festlegen.

Wenn eine andere Person angeklickt wird, wollen wir die vorher angezeigten Informationen verbergen. Dazu können wir eine Klasse einsetzen, um nachzuverfolgen, welche Informationen gerade auf dem Bildschirm angezeigt werden:

```
var $member = $(this);
if ($member.hasClass('active')) {
    return;
}
$('div.member.active')
    .removeClass('active')
    .children('div').fadeOut();
$member.addClass('active');
```

**Listing 11-7**

Dieser neue Code, der am Anfang von `showDetails()` platziert ist, fügt den angeklickten Personen die Klasse `active` hinzu. Indem wir diese Klasse suchen, können wir die sichtbaren Elemente einfach finden und sie ausblenden. Wir können die Klasse auch einsetzen, um zurückzukehren, ohne etwas zu unternehmen, wenn die angeklickte Person bereits aktiv ist.

Beachten Sie, dass unser `.fadeOut()`-Aufruf auch die schnellere Dauer von 250 Millisekunden verwendet, die wir definiert haben. Die Vorgaben gelten genauso für die mitgelieferten jQuery-Effekte wie für die eigenen `.animate()`-Aufrufe.

## 11.4 Easing mit mehreren Eigenschaften

ie `showDetails()`-Funktion liefert beinahe den Ausfaltungseffekt, den wir haben möchten, da aber die Eigenschaften `top` und `left` mit gleicher Rate animiert werden, wirkt sie eher wie ein Gleiteffekt. Wir können den Effekt geringfügig verändern, indem wir die Easing-Formel für die `top`-Eigenschaft in `easeInQuart` ändern, wodurch der Pfad gebogen ist und nicht gerade verläuft. Vergessen Sie dabei aber nicht, dass ein anderes Easing als `swing` oder `linear` ein Plug-in benötigt, wie den Effektkern von jQuery UI (<http://jqueryui.com/>) oder das eigenständige jQuery-Easing-Plug-in (<http://gsgd.co.uk/sandbox/jquery/easing/>):

```
$member.find('div').css({
    display: 'block',
    left: '-300px',
    top: 0
}).each(function(index) {
    $(this).animate({
        left: 0,
        top: 25 * index
    }, {
        duration: 'slow',
        specialEasing: {
            top: 'easeInQuart'
        }
    });
});
```

**Listing 11-8**

Die Option `specialEasing` bietet uns einen Satz unterschiedlicher Beschleunigungskurven für jede animierbare Eigenschaft. Alle Eigenschaften, die nicht in der Option enthalten sind, verwenden die der `easing`-Option mitgegebene Formel oder, wenn keine angegeben ist, als Voreinstellung die `swing`-Formel.

Jetzt haben wir für die meisten Informationen zu einer Person eine ansprechende Animation. Allerdings zeigen wir noch keinen Lebenslauf der Person an. Bevor wir das in Angriff nehmen, müssen wir noch einen kleinen Umweg über den verzögerten jQuery-Objektmechanismus machen.

## 11.5 Verzögerte Objekte

In jQuery 1.5 wurde in der Bibliothek ein Konzept namens *verzögertes Objekt* eingeführt. Ein verzögertes Objekt kapselt eine Operation, die Zeit zum Abschluss benötigt. Diese Objekte ermöglichen es uns, auf einfache Weise mit Situationen umzugehen, in denen wir erst reagieren wollen, wenn ein Prozess abgeschlossen ist, wir aber nicht genau wissen, wie lange er dauert oder ob er überhaupt fertig wird.

Ein neues verzögertes Objekt kann jederzeit durch Aufrufen des `$.Deferred()`-Konstruktors erzeugt werden. Wenn wir ein solches Objekt haben, können wir lange dauernde Operationen durchführen und dann die Methoden `.resolve()` oder `.reject()` aufrufen, um anzusehen, dass der Vorgang erfolgreich war oder nicht. Jedoch ist es etwas ungewöhnlich, dies manuell zu tun. Statt unsere verzögerten Objekte manuell zu erzeugen, erledigt dies jQuery oder seine Plug-ins und es kümmert sich auch um die Auflösung bzw. Ablehnung. Wir müssen einfach nur lernen, wie wir mit dem erzeugten Objekt umgehen.

Verzögerte Objekte zu erstellen ist ein sehr fortgeschrittenes Thema – fortgeschrittener als es dem Rahmen dieses Buches angemessen ist. Statt zu erklären, wie der `$.Deferred()`-Konstruktor funktioniert, beschränken wir uns hier darauf, welche jQuery-Effekte Nutzen aus verzögerten Objekten ziehen. In Kapitel 13 untersuchen wir diese Objekte weiter im Kontext mit Ajax-Aufrufen.

Jedes verzögerte Objekt gibt ein Versprechen, ein sogenanntes *Promise*, anderem Code Daten zu liefern. Dieses Versprechen wird durch ein anderes Objekt dargestellt inklusive eines eigenen Methodensatzes. Von jedem verzögerten Objekt können wir sein versprochenes Objekt durch Aufrufen der Methode `.promise()` erhalten. Dann können wir Methoden des `promise`-Objekts aufrufen, um Handler daran zu binden, die ausgeführt werden, wenn das Versprechen eingelöst wird:

- Die Methode `.done()` fügt einen Handler an, der aufgerufen wird, wenn das verzögerte Objekt erfolgreich aufgelöst wird.
- Die Methode `.fail()` fügt einen Handler an, der aufgerufen wird, wenn das verzögerte Objekt abgelehnt wird.
- Die Methode `.always()` fügt einen Handler an, der aufgerufen wird, wenn das verzögerte Objekt seine Aufgabe beendet hat, und zwar entweder durch Auflösung oder durch Ablehnung.

Diese Handler gleichen den Callbacks, die wir an `.bind()` liefern, da sie Funktionen sind, die bei einem Ereignis aufgerufen werden. Wir können auch mehrere Handler an dasselbe *Promise* anfügen und alle werden zum passenden Zeitpunkt aufgerufen. Trotzdem gibt es ein paar wichtige Unterschiede. *Promise*-Handler werden nur einmal aufgerufen, sodass das verzögerte Objekt kein zweites Mal aufgelöst werden kann. Ein *Promise*-Handler wird außerdem sofort aufgerufen, wenn das verzögerte Objekt bereits aufgelöst ist, wenn wir den Handler anfügen.

Jetzt können wir dieses leistungsfähige Werkzeug nutzen, indem wir eines der von jQuery für uns erstellten verzögerten Objekte untersuchen.

### 11.5.1 Animations-Promises

Jede jQuery-Sammlung besitzt einen Satz verzögerter Objekte, die den Status von Elementoperationen in Warteschlangen verfolgen. Durch Aufrufen der `.promise()`-Methode im jQuery-Objekt erhalten wir ein *Promise*-Objekt, das aufgelöst wird, wenn eine Warteschleife beendet wird. Wir können insbesondere dieses *Promise* einsetzen, um nach Abschluss aller laufenden Animationen Aktionen mit einem der passenden Elemente durchzuführen.

So, wie wir eine Funktion `showDetails()` dazu verwenden können, einen Personennamen und Adressinformationen anzuzeigen, so können wir eine Funktion `showBio()` schreiben, um biografische Informationen sichtbar zu machen. Zuerst müssen wir dem `<body>`-Element ein neues `<div>`-Element hinzufügen und eine Reihe von Map-Optionen setzen:

```
var $movable = $('

</div>')
    .appendTo('body');

var bioBaseStyles = {
    display: 'none',
    height: '5px',
    width: '25px'
},
    bioEffects = {
        duration: 800,
        easing: 'easeOutQuart',
        specialEasing: {
            opacity: 'linear'
        }
    };


```

**Listing 11-9**

Genau dieses neue, bewegliche `<div>`-Element wollen wir animieren, nachdem wir es mit einer Fassung des Lebenslaufs gespickt haben. Ein Wrapperelement wie dieses ist besonders nützlich, wenn wir die Breite und Höhe eines Elements animieren. Wir können seine Eigenschaft `overflow` auf `hidden` setzen und dann eine explizite Breite und Höhe für die Informationen darin definieren, um ein laufendes Umbrechen des Texts zu verhindern, das aufgetreten wäre, wenn wir stattdessen die `<div>`-Elemente des Lebenslaufs selbst animiert hätten.

Wir können die Funktion `showBio()` einsetzen, um zu entscheiden, wie die Start- und Endformate des sich bewegenden `<div>`-Elements abhängig von der angeklickten Person aussehen sollen. Beachten Sie, dass wir die `$.extend()`-

Methode verwenden, um den Satz von Basisformaten, die konstant bleiben, mit den `top`- und `left`-Eigenschaften, die sich abhängig von der Position der Person verändern, zusammenzuführen. Jetzt muss nur noch `.css()` zum Einsatz kommen, um die Startformate festzulegen, und `.animate()` für das Endformat:

```
function showBio() {
    var $member = $(this).parent(),
        $bio = $member.find('p.bio'),
        startStyles = $.extend(bioBaseStyles, $member.offset()),
        endStyles = {
            width: $bio.width(),
            top: $member.offset().top + 5,
            left: $member.width() + $member.offset().left - 5,
            opacity: 'show'
        };
    $movable
        .html($bio.clone())
        .css(startStyles)
        .animate(endStyles, bioEffects)
        .animate({height: $bio.height()}, {
            easing: 'easeOutQuart'
        });
}
```

**Listing 11-10**

Wir stellen zwei `.animate()`-Methoden in die Warteschlange, sodass der Lebenslauf erst von links kommt, dann breiter wird und komplett undurchsichtig und dann nach unten auf seine volle Höhe gleitet, wenn er die endgültige Position erreicht hat.

Wir haben in Kapitel 4 besprochen, dass Callback-Funktionen in jQuery-Animationsmethoden aufgerufen werden, wenn die Animation für *jedes Element in der Sammlung* abgeschlossen ist. Wir möchten den Lebenslauf der Person anzeigen, nachdem die anderen `<div>`-Elemente erscheinen. Bevor jQuery die `.promise()`-Methode eingeführt hat, wäre das eine mühselige Aufgabe, bei der wir bei jedem Callback von der Gesamtanzahl von Elementen rückwärts zählen müssten bis zum letzten Mal, wo wir den Code zur Animation des Lebenslaufs ausgeführt hätten. Jetzt können wir einfach die `.promise()`- und `.done()`-Methoden mit der `.each()`-Methode innerhalb unserer `showDetails()`-Funktion verketten:

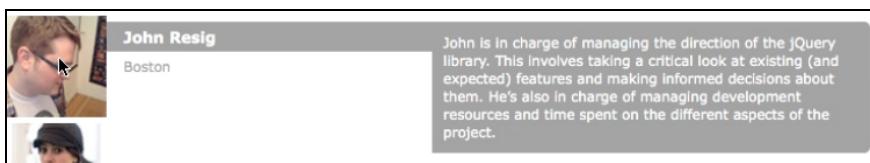
```
function showDetails() {
    var $member = $(this).parent();
    if ($member.hasClass('active')) {
        return;
    }
    $movable.fadeOut();

    $('div.member.active')
        .removeClass('active')
        .children('div').fadeOut();

    $member.addClass('active');
    $member.find('div').css({
        display: 'block',
        left: '-300px',
        top: 0
    }).each(function(index) {
        $(this).animate({
            left: 0,
            top: 25 * index
        }, {
            duration: 'slow',
            specialEasing: {
                top: 'easeInQuart'
            }
        });
    }).promise().done(showBio);
}
```

**Listing 11-11**

Die Methode `.done()` nimmt Bezug auf unsere Funktion `showBio()` in ihrem Argument. Ein Klick auf das Bild bringt jetzt die Informationen dieser Person mit einer ansprechenden Animation zur Ansicht, wie die folgenden Screenshots zeigen:



Beachten Sie auch, dass wir `$movable.fadeOut()` oben in der Funktion untergebracht haben. Das hat zwar beim ersten Aufruf von `showDetails()` keine Auswirkung, aber in den folgenden Aufrufen blendet es die momentan angezeigte Biografie aus, bevor die neuen Informationen per Animation sichtbar werden.

## 11.6 Zusammenfassung

In diesem Kapitel haben wir verschiedene Techniken näher untersucht, die uns bei der Erstellung schöner Animationen unterstützen, die unsere Benutzer erfreuen. Wir können jetzt die Beschleunigung und Verlangsamung jeder animierten Eigenschaft steuern und sie individuell oder global anhalten, wenn es notwendig ist. Wir haben die Eigenschaften der Effektbibliothek von jQuery kennengelernt und wissen nun, wie wir einige davon an unsere Bedürfnisse anpassen können. Außerdem haben wir einen ersten Ausflug in das jQuery-System für verzögerte Objekte unternommen, das wir in Kapitel 13 weiter erforschen werden.

### 11.6.1 Literatur

Eine vollständige Liste der verfügbaren Effekte und Animationsmethoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter folgender URL: <http://api.jquery.com/>.

## 11.7 Übungsaufgaben

Um die folgenden Übungsaufgaben durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Definieren Sie eine neue Geschwindigkeitskonstante für Animationen namens `zippy` und wenden Sie sie auf den Anzeigeeffekt für die Lebensläufe an.
2. Ändern Sie das Easing der Horizontalbewegung der Personeninformationen, sodass sie in ihre Position springen.
3. Fügen Sie dem *Promise* eine zweite, verzögerte Callback-Funktion hinzu, die der Position des `<div>`-Elements der aktuellen Person eine `highlight`-Klasse hinzufügt.
4. *Schwierig:* Fügen Sie eine zweisekündige Verzögerung ein, bevor der Lebenslauf angezeigt wird. Verwenden Sie dazu die jQuery-Methode `.delay()`.
5. *Schwierig:* Wenn auf das aktive Foto geklickt wird, sollen die biografischen Informationen verschwinden. Halten Sie zuvor alle laufenden Animationen an.



## 12 DOM-Manipulation für Fortgeschrittene

In Kapitel 5 haben wir eine Einführung in die leistungsfähigen Funktionen erhalten, um den Inhalt des DOM zu manipulieren. Wir kennen nun verschiedene Möglichkeiten, neuen Inhalt einzufügen, Inhalt zu verschieben oder aus dem Dokument zu entfernen. Wir wissen auch, wie wir Attribute und Eigenschaften von Elementen nach unseren Wünschen anpassen können.

In diesem Kapitel werden wir tiefer in die jQuery-Funktionen zur Manipulation von DOM-Objekten einsteigen und weiterführende Verfahren kennenlernen. Wir sehen uns an, wie wir Daten über DOM-Elemente speichern und wieder abrufen und wie wir damit Seiteninhalte bei Bedarf wieder aufbauen. Indem wir die Funktionsweise der jQuery-Funktionen für das DOM besser verstehen, lernen wir auch, wie wir die Leistung unseres Codes verbessern und die Bibliothek verändern können, um erweiterte CSS-Funktionen zu aktivieren.

### 12.1 Tabellenzeilen sortieren

Die Mehrzahl der in diesem Kapitel angesprochenen Themen kann anhand der Sortierung von Tabellenzeilen erläutert werden. Diese häufige Aufgabe hilft Benutzern, gewünschte Funktionen schnell zu finden. Es gibt natürlich verschiedene Möglichkeiten, wie wir das bewerkstelligen können.

#### 12.1.1 Serverseitiges Sortieren

Ein häufig benutzerter Ansatz ist, die Daten auf dem Server zu sortieren. Daten werden oft durch eine Datenbank bereitgestellt, was bedeutet, dass der Code die Informationen von dort in einer bestimmten Ordnung abruft (z.B. mit der SQL-Klausel `ORDER BY`). Wenn wir serverseitigen Code einsetzen können, ist es die einfachste Lösung, mit einer vernünftigen Standardsortierung zu beginnen.

Die Sortierung ist jedoch dann am nützlichsten, wenn der Benutzer sie selbst festlegen kann. Eine übliche Benutzerschnittstelle besteht darin, die sortierbaren

Spalten im Tabellenkopf (`<th>`) als Links darzustellen. Diese Links führen auf die aktuelle Seite, enthalten aber einen Abfragestring, der angibt, wie die Spalten zu sortieren sind:

```
<table id="my-data">
    <thead>
        <tr>
            <th class="name">
                <a href="index.php?sort=name">Name</a>
            </th>
            <th class="date">
                <a href="index.php?sort=date">Date</a>
            </th>
        </tr>
    </thead>
    <tbody>
        ...
    </tbody>
</table>
```

Der Server kann auf den Abfragestring reagieren und den Inhalt der Datenbank in einer anderen Sortierung liefern.

### 12.1.2 Sortierung mit Ajax

Dieses Verfahren ist einfach, hat jedoch ein Neuladen der Seite für jeden Sortierungsvorgang zur Folge. Wie wir gesehen haben, können wir mit jQuery das Neuladen solcher Seiten unterbinden, indem wir Ajax-Methoden einsetzen. Wenn wir die Kopfzeilen wie gezeigt als Links angelegt haben, können wir jQuery-Code verwenden, um diese Links in Ajax-Aufrufe umzuändern:

```
$(document).ready(function() {
    $('#my-data th a').click(function() {
        $('#my-data tbody').load($(this).attr('href'));
        return false;
    });
});
```

Wenn die Links jetzt angeklickt werden, sendet jQuery eine Ajax-Anfrage für dieselbe Seite an den Server. Wenn jQuery verwendet wird, um mittels Ajax eine Anfrage zu senden, setzt es den HTTP-Header `X-Requested-With` auf `XMLHttpRequest`, sodass der Server erkennen kann, dass es sich um eine Ajax-Anfrage handelt. Der serverseitige Code kann so geschrieben werden, dass er nur den Inhalt des `<tbody>`-Elements zurücksendet und nicht die vollständige Seite, wenn dieser Parameter gesetzt ist. Jetzt können wir die gesendeten Daten mit dem Inhalt des `<tbody>`-Elements austauschen.

Es handelt sich hier um ein Beispiel für fortschreitende Verbesserung. Die Seite funktioniert ohne JavaScript ganz hervorragend, weil die Links für die serverseitige Sortierung immer noch vorhanden sind. Wenn JavaScript jedoch verfügbar ist, übernimmt Ajax die Seitenanfrage und ermöglicht uns die Sortierung, ohne die Seite komplett neu laden zu müssen.

### 12.1.3 Sortierung mit JavaScript

Es gibt jedoch Fälle, in denen wir nicht auf eine Serverantwort warten möchten oder keine Sprache zur Verfügung haben, die serverseitiges Scripting ermöglicht. Dann ist es eine gute Alternative, die Sortierung vollständig im Browser mithilfe von clientseitigem Scripting und JavaScript zu erledigen.

Um die verschiedenen Sortierverfahren in diesem Kapitel zu erläutern, richten wir drei verschiedene JavaScript-Sortiermechanismen ein. Jeder erfüllt dieselbe Aufgabe, aber auf unterschiedliche Weise. Die von uns sortierten Tabellen haben unterschiedliche HTML-Strukturen, um die verschiedenen JavaScript-Techniken zu veranschaulichen. Jede enthält jedoch Spalten mit Buchtiteln, Autorennamen, Erscheinungsdatum und Preis. Die erste Tabelle hat folgende einfache Struktur:

```
<table id="t-1" class="sortable">
  <thead>
    <tr>
      <th></th>
      <th class="sort-alpha">Title</th>
      <th class="sort-alpha">Author(s)</th>
      <th class="sort-date">Publish Date</th>
      <th class="sort-numeric">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td></td>
      <td>Drupal 7</td>
      <td>David <span class="sort-key">Mercer</span></td>
      <td>September 2010</td>
      <td>$44.99</td>
    </tr>
    <!-- Code wird fortgesetzt -->
  </tbody>
</table>
```

Bevor wir die Tabelle mit JavaScript erweitern, sehen die ersten Zeilen wie der folgende Screenshot aus:

Title	Author(s)	Publish Date	Price
 Drupal 7	David Mercer	September 2010	\$44.99
 Amazon SimpleDB: LITE	Prabhakar Chaganti, Rich Helms	May 2011	\$9.99
 Object-Oriented JavaScript	Stoyan Stefanov	July 2008	\$39.99

## 12.2 Elemente verschieben und einfügen – Teil 2

In den kommenden Beispielen erstellen wir einen flexiblen Sortiermechanismus, der mit jeder einzelnen Spalte funktioniert. Dazu verwenden wir jQuery-Methoden zur DOM-Manipulation, um neue Elemente einzufügen und andere, bestehende Elemente an andere Stellen zu verschieben. Wir beginnen mit dem einfachsten Teil des Puzzles: der Verlinkung der Tabellenköpfe.

### 12.2.1 Links um bestehenden Text herum einfügen

Wir möchten die Tabellenköpfe in Links umwandeln, die die Daten nach ihrer jeweiligen Spalte sortieren. Mit der jQuery-Methode `.wrapInner()` können wir sie hinzufügen. Wie wir noch aus Kapitel 5 wissen, platziert `.wrapInner()` ein neues Element (in diesem Falle ein `<a>`) innerhalb eines passenden Elements, aber um die Kindelemente herum:

```
$(document).ready(function() {
    var $table1 = $('#t-1');
    var $headers = $table1.find('thead th').slice(1);
    $headers
        .wrapInner('<a href="#"></a>')
        .addClass('sort');
});
```

#### *Listing 12-1*

Wir haben das erste `<th>` jeder Tabelle übersprungen (mittels `.slice()`), da es keinen Text außer Leerraum enthält und es nicht notwendig ist, die Umschlagfotos zu benennen oder zu sortieren. Wir haben den verbleibenden `<th>`-Elementen die

Klasse `sort` zugewiesen, sodass wir sie in CSS von den nicht sortierbaren Elementen unterscheiden können. Jetzt sehen die Kopfzeilen wie der folgende Screenshot aus:

	◆ Title		◆ Author(s)		◆ Publish Date		◆ Price
	Drupal 7		David Mercer		September 2010		\$44.99
	Amazon SimpleDB: LITE		Prabhakar Chaganti, Rich Helms		May 2011		\$9.99
	Object-Oriented JavaScript		Stoyan Stefanov		July 2008		\$39.99

Dies ist ein Beispiel für das Gegenstück der fortschreitenden Verbesserung, die allmähliche Funktionsminderung. Im Gegensatz zur Lösung mit Ajax funktioniert diese Funktion nicht ohne JavaScript. In diesem Beispiel gehen wir davon aus, dass der Server über keine Skriptsprache verfügt. Da JavaScript für die Sortierung notwendig ist, fügen wie die Klasse `sort` und die Ankerpunkte im gesamten Code hinzu, um klarzumachen, dass eine Sortierung nur möglich ist, wenn das Skript laufen kann. Und da wir gerade schon Links erstellen, unterstützen wir Benutzer, die sich mit der Tastatur (mittels der Tabulatoraste) durch die Tabelle bewegen wollen gleich mit, statt nur einen optischen Hinweis auf die Sortiermöglichkeit anzubringen. Die Seite verschlechtert sich dahingehend, dass sie zwar angezeigt wird, jedoch keine Sortierung mehr möglich ist.

### 12.2.2 Einfache JavaScript-Arrays sortieren

Um den Sortiervorgang auszuführen, nutzen wir die in JavaScript integrierte Methode `.sort()`. Sie führt eine Sortierung im Array durch und verwendet die Komparatorfunktion als Argument. Diese Funktion vergleicht zwei Elemente im Array und gibt eine positive oder negative Zahl zurück, abhängig davon, welches Element im sortierten Array zuerst vorkommt.

Nehmen wir ein einfaches Array wie dieses hier:

```
var arr = [52, 97, 3, 62, 10, 63, 64, 1, 9, 3, 4];
```

Wir können dieses Array durch Aufruf von `arr.sort()` sortieren. Danach sind die Elemente wie folgt angeordnet:

```
[1, 10, 3, 3, 4, 52, 62, 63, 64, 9, 97]
```

Per Voreinstellung werden die Elemente *lexikografisch* sortiert (in alphabetischer Reihenfolge). In diesem Fall wäre es jedoch sinnvoller, sie *numerisch* zu sortieren. Dazu müssen wir der Methode `.sort()` eine Komparatorfunktion mitgeben:

```
arr.sort(function(a,b) {  
    if (a < b) {  
        return -1;  
    }  
    if (a > b) {  
        return 1;  
    }  
    return 0;  
});
```

Diese Funktion gibt eine negative Zahl zurück, wenn a zuerst im sortierten Array vorkommt, eine positive Zahl, wenn b zuerst kommt, und null, wenn die Reihenfolge der Elemente nicht relevant ist. Mit diesen Informationen kann die `.sort()`-Methode die Elemente richtig sortieren:

```
[1, 3, 3, 4, 9, 10, 52, 62, 63, 64, 97]
```

Nun werden wir die `.sort()`-Methode auf unsere Tabellenzeilen anwenden.

### 12.2.3 DOM-Elemente sortieren

Lassen Sie uns jetzt die Titelspalte der Tabelle sortieren. Auch wenn wir ihr und den anderen Spalten die Klasse `sort` hinzugefügt haben, besitzt die Titelzelle bereits eine `sort-alpha`-Klasse, die sich im HTML befand. Die anderen Kopfzellen sind ähnlich behandelt worden, abhängig von der Art ihrer Sortierung. Jetzt werden wir uns auf den Titelkopf konzentrieren, der eine einfache alphabetische Sortierung benötigt.

```
$headers.bind('click', function(event) {  
    event.preventDefault();  
    var column = $(this).index();  
    var rows = $table1.find('tbody > tr').get();  
    rows.sort(function(a, b) {  
        var keyA = $(a).children('td').eq(column).text();  
        keyA = $.trim(keyA).toUpperCase();  
        var keyB = $(b).children('td').eq(column).text();  
        keyB = $.trim(keyB).toUpperCase();  
        if (keyA < keyB) return -1;  
    });  
    $table1[0].tbody.replaceChild(rows[0], rows[rows.length - 1]);  
    $table1.trigger('sort');  
});
```

```
    if (keyA > keyB) return 1;
    return 0;
});

$.each(rows, function(index, row) {
    $table1.children('tbody').append(row);
});
});
```

**Listing 12-2**

Nachdem wir jetzt den Index der angeklickten Kopfzelle ermittelt haben, können wir ein Array mit allen Datenzeilen abrufen. Dies ist ein hervorragendes Beispiel, wie aus einem jQuery-Objekt mittels `.get()` ein Array mit DOM-Knoten erstellt werden kann. Auch wenn sich jQuery-Objekte in vielen Details wie Arrays verhalten, verfügen sie nicht über die nativen Array-Methoden wie `.pop()` oder `.shift()`.

Intern definiert jQuery tatsächlich einige Methoden, die sich wie native Array-Methoden verhalten. In Version 1.6 gibt es beispielsweise mit `.sort()`, `.push()` und `.splice()` Methoden für jQuery-Objekte. Da es sich jedoch um nicht öffentlich dokumentierte interne Methoden handelt, können wir uns nicht darauf verlassen, dass sie sich in unserem Code wie erwartet verhalten, und sollten daher vermeiden, sie für jQuery-Objekte zu verwenden.

Nachdem wir jetzt ein Array mit DOM-Knoten haben, können wir sie sortieren, dazu müssen wir jedoch eine passende *Komparatorfunktion* schreiben. Wir werden die Zeilen entsprechend dem Inhalt der relevanten Tabellenzellen sortieren, sodass dies die Informationen sind, die die Komparatorfunktion untersuchen muss. Wir wissen, welche Zellen untersucht werden sollen, da wir mit dem `.index()`-Aufruf den Spaltenindex erfasst haben. Mit der jQuery-Funktion `$.trim()` werden wir führende und nachfolgende Leerstellen entfernen und dann konvertieren wir den Text in Großbuchstaben, da Stringvergleiche in JavaScript Schreibweisenabhängig (case-sensitiv) sind, während unsere Sortierung unabhängig von der Schreibweise sein soll. Wir legen die Schlüsselwerte in Variablen ab, um überflüssige Berechnungen zu unterbinden, vergleichen sie, und geben wie vorher erläutert einen positiven oder negativen Wert zurück.

Jetzt ist unser Array sortiert, das DOM ist dadurch jedoch noch nicht verändert worden. Um die Zeilen umzustellen, müssen wir DOM-Manipulationsmethoden aufrufen. Dies führen wir zeilenweise durch und fügen jede Zeile neu ein, während die Schleife darüberläuft. Da `.append()` keine Knoten klonen, werden sie verschoben und nicht kopiert. Unsere Tabelle ist jetzt wie folgt sortiert:

◀ Title	◀ Author(s)	◀ Publish Date	◀ Price
	Amazon SimpleDB: LITE Prabhakar Chaganti, Rich Helms	May 2011	\$9.99
	CakePHP 1.3 Application Development Cookbook Mariano Iglesias	March 2011	\$39.99
	Cocoa and Objective-C Cookbook Jeff Hawkins	May 2011	\$39.99

## 12.3 Daten zusammen mit DOM-Elementen ablegen

Unser Code läuft jetzt, ist aber etwas langsam. Schuld daran ist die Komparatorfunktion, die sehr viel Arbeit leisten muss. Der Komparator wird während des Sortiervorgangs häufig aufgerufen, was dazu führt, dass sich jeder Zeitverlust vervielfacht.

### Performance beim Array-Sortieren

Der aktuelle Sortieralgorithmus, den JavaScript benutzt, folgt keinem bestimmten Standard. Er kann eine einfache Sortierung wie *Bubblesort* sein (analytisch gesehen die langsamste Vorgehensweise, nämlich  $\Theta(n^2)$ ) oder ein weiterentwickelter Ansatz wie *Quicksort* (das durchschnittlich  $\Theta(n \log n)$  dauert). Trotzdem kann man behaupten, dass eine Verdoppelung der Elemente im Array zu mehr als einer Verdoppelung der Aufrufe der Komparatorfunktion führt.

Eine Abhilfe für unseren langsamen Komparator besteht in der Vorberechnung der Schlüssel für den Vergleich. Wir können die rechenaufwendigen Arbeiten in einer vorgesetzten Schleife erledigen und das Ergebnis mit der jQuery-Methode `.data()` ablegen, die beliebige, den Seitenelementen zugehörige Informationen speichert. Dann können wir die Schlüssel in der Komparatorfunktion einfach untersuchen und die Sortierung läuft merklich schneller ab:

```
var rows = $table1.find('tbody > tr').each(function() {  
    var key = $(this).children('td').eq(column).text();  
    $(this).data('sortKey', $.trim(key).toUpperCase());  
});get();
```

```
rows.sort(function(a, b) {
    var keyA = $(a).data('sortKey');
    var keyB = $(b).data('sortKey');
    if (keyA < keyB) return -1;
    if (keyA > keyB) return 1;
    return 0;
});
```

**Listing 12–3**

Die `.data()`-Methode bietet zusammen mit ihrer Ergänzung `.removeData()` einen Speichermechanismus, der eine bequeme Alternative zu expando-Eigenschaften oder nicht standardisierten Eigenschaften ist, die DOM-Elementen direkt angefügt werden. Indem wir `.data()` statt expando-Eigenschaften verwenden, vermeiden wir potenzielle Probleme mit Speicherlecks im Internet Explorer.

### 12.3.1 Zusätzliche Vorberechnungen

Jetzt möchten wir die gleiche Art Sortierung für die Tabellenspalte *Author(s)* einsetzen. Da der Tabellenkopf die Klasse `sort-alpha` hat, kann die *Author(s)*-Spalte mit unserem vorhandenen Code sortiert werden. Idealerweise sollten die Autoren jedoch nicht nach ihrem Vornamen, sondern nach dem Nachnamen sortiert werden. Da manche Bücher mehrere Autoren haben und manche Autoren zwei Vornamen oder eine Mittelinitiale, benötigen wir eine Hilfestellung, um zu entscheiden, welcher Textteil als Sortierschlüssel verwendet werden soll. Diese Hilfestellung erhalten wir, indem wir den relevanten Zellenteil in einem Tag verpacken:

```
<td>David <span class="sort-key">Mercer</span></td>
```

Jetzt müssen wir unseren Sortiercode so anpassen, dass dieses Tag berücksichtigt wird, ohne das bestehende Verhalten für die Titelspalte zu verändern, denn das funktioniert ja bereits. Indem wir den gekennzeichneten Sortierschlüssel an den bereits berechneten Schlüssel anfügen, können wir anhand des Nachnamens sortieren, wenn dieser aufgerufen wird, haben aber den gesamten String als Fallback:

```
var rows = $table1.find('tbody > tr').each(function() {
    var $cell = $(this).children('td').eq(column);
    var key = $cell.find('span.sort-key').text() + ' ';
    key += $.trim($cell.text()).toUpperCase();
    $(this).data('sortKey', key);
}).get();
```

**Listing 12–4**

Die Sortierung nach der *Author(s)*-Spalte nutzt jetzt den gelieferten Schlüssel und sortiert nach dem Nachnamen:

◆ Title	◆ Author(s)	◆ Publish Date	◆ Price
	WordPress 3 Plugin Development Essentials	Brian Bondari, Everett Griffiths	March 2011 \$39.99
	Magento 1.4 Themes Design	Richard Carter	January 2011 \$39.99
	Amazon SimpleDB: LITE	Prabhakar Chaganti, Rich Helms	May 2011 \$9.99

Sind zwei Nachnamen identisch, verwendet die Sortierung den gesamten String als Entscheidungshilfe für die Positionierung.

### 12.3.2 Nicht-String-Daten speichern

Unser Benutzer sollte in der Lage sei, nicht nur nach den Spalten *Title* und *Author(s)* zu sortieren, sondern auch nach *Publish Date* und *Price*. Da wir unsere Komparatorfunktion optimiert haben, kann sie alle Arten von Daten verarbeiten, aber zuerst müssen die berechneten Schlüssel für andere Datentypen angepasst werden. Im Fall von Preisen müssen wir z.B. ein führendes \$- oder €-Zeichen entfernen und den Rest so verarbeiten, dass wir ihn numerisch vergleichen können:

```
var key = parseFloat($cell.text().replace(/^\d.*/, ''));  
if (isNaN(key)) {  
    key = 0;  
}
```

Der hier verwendete reguläre Ausdruck entfernt alle führenden Zeichen, die keine Zahlen und Dezimalpunkte sind, und gibt das Ergebnis an `parseFloat()`. Das Ergebnis von `parseFloat()` muss wiederum geprüft werden, denn wenn der Text nicht in eine Zahl umgewandelt werden kann, wird `Nan` (Not a Number) zurückgegeben. Das kann sich verheerend auf `.sort()` auswirken, weshalb wir alle `Nan` auf 0 setzen.

Für die Datumszellen verwenden wir das JavaScript-Objekt `Date`:

```
var key = Date.parse('1 ' + $cell.text());
```

Die Daten in dieser Tabelle enthalten nur Monat und Jahr. `.DateParse()` benötigt jedoch ein vollständiges Datum, weshalb wir dem String 1 voranstellen. So erhalten wir auch eine Tagesangabe, die in einen Timestamp umgewandelt werden kann, der sich mit dem normalen Komparator sortieren lässt.

Wir können diese Ausdrücke über verschiedene Funktionen aufteilen, sodass wir später die für die jeweilige Klasse des Tabellenkopfs richtige aufrufen können.

```
$headers
.each(function() {
    var keyType = this.className.replace(/^sort-/, '');
    $(this).data('keyType', keyType);
})
.wrapInner('<a href="#"></a>')
.addClass('sort');

var sortKeys = {
    alpha: function($cell) {
        var key = $cell.find('span.sort-key').text() + ' ';
        key += $.trim($cell.text()).toUpperCase();
        return key;
    },
    numeric: function($cell) {
        var num = $cell.text().replace(/^[^\d.]*/, '');
        var key = parseFloat(num);
        if (isNaN(key)) {
            key = 0;
        }
        return key;
    },
    date: function($cell) {
        var key = Date.parse('1 ' + $cell.text());
        return key;
    }
};
```

**Listing 12–5**

Wir haben das Skript ein wenig modifiziert, um die `keyType`-Daten jedes Spaltenkopfs entsprechend seinem Klassennamen zu speichern, bevor wir die Klasse `sort` hinzufügen. Wir entfernen dann den `sort`-Anteil der Klasse, sodass wir `alpha`, `numeric` oder `date` erhalten. Indem wir aus jeder Sortierfunktion eine Methode der `sortKeys`-Map machen, können wir die Array-Schreibweise beibehalten und den Wert des `keyType` der Kopzfzelle im entsprechenden Funktionsaufruf verwenden.

Normalerweise verwenden wir für Methodenaufrufe die Punktnotation. So rufen wir in diesem Buch alle Methoden von jQuery-Objekten auf. Um beispielsweise `<div class="foo">` die Klasse `bar` zuzuweisen, schreiben wir `$(‘div.foo’).addClass(‘bar’)`. Da JavaScript es erlaubt, Eigenschaften und Methoden entweder in Punkt- oder Array-Schreibweise anzugeben, könnten wir auch `$(‘div.foo’)[‘addClass’](‘bar’)` schreiben. Meistens bringt diese Schreibweise keinen großen Vorteil, sie kann jedoch für den gelegentlichen Aufruf von Methoden ohne zahlreiche `if`-Anweisungen nützlich sein. Für unsere `sortKeys`-Map könnten wir die Methode `alpha` mittels `sortKeys.alpha($cell)` oder `sortKeys[‘alpha’]($cell)` aufrufen

oder, wenn der Methodenname in einer `keyType`-Variablen gespeichert ist, als `sortKeys[keyType]($cell)`. Wir verwenden die dritte Variante im `click`-Handler, wie hier:

```
var $header = $(this),
    column = $header.index(),
    keyCode = $header.data('keyCode');

if ( !$.isFunction(sortKeys[keyCode]) ) {
    return;
}

var rows = $table1.find('tbody > tr').each(function() {
    var $cell = $(this).children('td').eq(column);
        $cell.data('sortKey', sortKeys[keyCode]($cell));
}).get();
```

**Listing 12–6**

Um ganz sicher zu gehen und JavaScript-Fehler zu vermeiden, haben wir auch dafür gesorgt, dass die Methode `sortKeys[keyType]` existiert, bevor wir weitermachen. Jetzt können wir auch nach Datum und Preis sortieren:

 <a href="#">Title</a>	 <a href="#">Author(s)</a>	 <a href="#">Publish Date</a>	 <a href="#">Price</a>
 Object-Oriented JavaScript	Stoyan Stefanov	July 2008	\$39.99
 jQuery 1.4 Reference Guide	Karl Swedberg, Jonathan Chaffer	January 2010	\$39.99
 Drupal 7	David Mercer	September 2010	\$44.99

### 12.3.3 Umkehren der Sortierrichtung

Unsere letzte Verbesserung der Sortierung ist die auf- und absteigende Reihenfolge. Wenn der Anwender auf eine Spalte klickt, die bereits ausgewählt ist, kehren wir die Sortierreihenfolge um.

Um die Sortierung umzukehren, müssen wir nur die vom Komparator zurückgegebenen Werte invertieren. Dies erreichen wir mit einer einfachen Variablen:

```
if (keyA < keyB) return -sortDirection;
if (keyA > keyB) return sortDirection;
return 0;
```

Wenn sortDirection gleich 1 ist, gleicht die Sortierung der vorherigen. Ist der Wert -1, wird die Sortierung umgekehrt. Dieses Konzept lässt sich leicht mit einigen Klassen kombinieren, um die bestehende Sortierreihenfolge zu berücksichtigen:

```
$headers.bind('click', function(event) {
    event.preventDefault();
    var $header = $(this),
        column = $header.index(),
        keyType = $header.data('keyType'),
        sortDirection = 1;

    if ( !$.isFunction(sortKeys[keyType]) ) {
        return;
    }

    if ($header.hasClass('sorted-asc')) {
        sortDirection = -1;
    }

    var rows = $table1.find('tbody > tr').each(function() {
        var $cell = $(this).children('td').eq(column);
        $(this).data('sortKey', sortKeys[keyType]($cell));
    }).get();

    rows.sort(function(a, b) {
        var keyA = $(a).data('sortKey');
        var keyB = $(b).data('sortKey');
        if (keyA < keyB) return -sortDirection;
        if (keyA > keyB) return sortDirection;
        return 0;
    });

    $headers.removeClass('sorted-asc sorted-desc');
    $header.addClass(sortDirection == 1 ? 'sorted-asc' : 'sorteddesc');

    $.each(rows, function(index, row) {
        $table1.children('tbody').append(row);
    });
});
```

**Listing 12-7**

Ein angenehmer Nebeneffekt beim Speichern der Klassen für die Sortierrichtung besteht darin, dass wir die Spaltenköpfe umgestalten können, um die aktuelle Reihenfolge anzuzeigen:

♦ Title	♦ Author(s)	♦ Publish Date	▼ Price
	Amazon SimpleDB: LITE Prabhakar Chaganti, Rich Helms	May 2011	\$9.99
	Object-Oriented JavaScript Stoyan Stefanov	July 2008	\$39.99
	jQuery 1.4 Reference Guide Karl Swedberg, Jonathan Chaffer	January 2010	\$39.99

## 12.4 HTML5 mit eigenen Datenattributen einsetzen

Bislang haben wir uns auf den Inhalt in den Tabellenzellen verlassen, um die Sortierreihenfolge zu bestimmen. Während wir die Zeilen dadurch korrekt sortieren konnten, dass wir den Inhalt verändert haben, können wir unseren Code effizienter gestalten, indem wir mehr HTML vom Server in Form von data\*-Attributten aus HTML5 ausgeben lassen. Die zweite Tabelle in unserem Beispiel enthält die folgenden Attribute:

```
<h3>Table 2</h3>
<table id="t-2" class="sortable">
  <thead>
    <tr>
      <th></th>
      <th data-sort='{"key":"title"}'>Title</th>
      <th data-sort='{"key":"authors"}'>Author(s)</th>
      <th data-sort='{"key":"publishedYM"}'>Publish Date</th>
      <th data-sort='{"key":"price"}'>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr data-book='{"img":"2862_OS.jpg","title":"DRUPAL 7",
      "authors":"MERCER DAVID","published":"September 2010",
      "price":44.99,"publishedYM":"2010-09"}'>
      <td></td>
      <td>Drupal 7</td>
      <td>David Mercer</td>
      <td>September 2010</td>
      <td>$44.99</td>
    </tr>
    <!-- Code wird fortgesetzt -->
  </tbody>
</table>
```

Beachten Sie, dass jedes `<th>`-Element (mit Ausnahme des ersten) ein `data-sort`-Attribut besitzt und jedes `<tr>`-Element ein `data-book`-Attribut. Wenn wir die Methode `.data()` verwenden, um den Wert eines Datenattributs zu bestimmen, konvertiert jQuery den Wert in eine Zahl, ein Array, Objekt, booleschen Wert oder null, wenn es einen dieser Typen erkennt. Für die Elemente `<th>` und `<tr>` interpretiert jQuery diese Werte als Objekte.

Damit jQuery den Wert eines Datenattributs in ein Objekt konvertiert, muss der String ein gültiges JSON-Format verwenden. Daher schreiben wir den Wert in einfachen Anführungszeichen und stellen jeden Schlüssel und String in doppelte Anführungszeichen (eigentlich hat der serverseitige Code ein assoziatives Array ins JSON-Format umcodiert). Um den Wert abzurufen, übergeben wir den Teil des Attributnamens nach `data-` an die Methode `.data()`. Zum Beispiel schreiben wir `($('th').first().data('sort'))`, um den Wert des `data-sort`-Attributs des ersten `<th>`-Elements zu erhalten. Um ganz gezielt den Wert der `key`-Eigenschaften von `("title")` zu bekommen, schreiben wir `($('th').first().data('sort').key)`. Nachdem ein eigenes Datenattribut auf diese Weise abgerufen wurde, werden die Daten intern durch jQuery gespeichert und das HTML `data*-Attribut` wird nicht wieder abgerufen oder verändert.

Datenattribute haben den großen Vorteil, dass die Werte separat vom Tabelleninhalt ausgegeben werden können. All die Arbeit, die wir in der ersten Tabelle aufwenden mussten, um die Sortierung zu verfeinern – Stringkonvertierung in Großbuchstaben, ändern des Datumsformats, Konvertierung des Preises in eine Zahl –, wurde schon für uns erledigt. Dadurch können wir wesentlich einfacheren und effizienteren Sortierungscode schreiben:

```
$document.ready(function() {
    var $table2 = $('#t-2');
    var $headers = $table2.find('thead th').slice(1);
    $headers
        .wrapInner('<a href="#"></a>')
        .addClass('sort');

    var rows = $table2.find('tbody > tr').get();

    $headers.bind('click', function(event) {
        event.preventDefault();
        var $header = $(this),
            sortKey = $header.data('sort').key,
            sortDirection = 1;

        if ($header.hasClass('sorted-asc')) {
            sortDirection = -1;
        }

        rows.sort(function(a, b) {
            var keyA = $(a).data('book')[sortKey];
            var keyB = $(b).data('book')[sortKey];
            if (sortKey === 'title') {
                if (sortDirection === 1) {
                    return keyA.localeCompare(keyB);
                } else {
                    return keyB.localeCompare(keyA);
                }
            } else {
                if (sortDirection === 1) {
                    return keyA - keyB;
                } else {
                    return keyB - keyA;
                }
            }
        });
    });
});
```

```
        if (keyA < keyB) return -sortDirection;
        if (keyA > keyB) return sortDirection;
        return 0;
    });

$headers.removeClass('sorted-asc sorted-desc');
$header.addClass(sortDirection == 1 ? 'sorted-asc' : 'sorted-desc');

$.each(rows, function(index, row) {
    $table2.children('tbody').append(row);
});
});

});
```

**Listing 12-8**

Die Einfachheit dieses Ansatzes ist klar: Die Variable `sortKey` wird mit `$header.data('sort').key` gesetzt und dann verwendet, um die Zeilenwerte mit `$(a).data('book')[sortKey]` und `$(b).data('book')[sortKey]` zu vergleichen. Die Effizienz besteht darin, dass keine Schleife über die Tabellenzeilen notwendig ist, in der eine der `sortKeys`-Funktionen vor Aufruf der `sort`-Funktion aufgerufen wird. Durch diese Kombination von Einfachheit und Effizienz haben wir auch die Performance des Codes verbessert und können ihn leichter modifizieren.

## 12.5 Zeilen mit JSON sortieren und erzeugen

Bis jetzt haben wir uns in diesem Kapitel dahin bewegt, mehr und mehr Informationen vom Server als HTML auszugeben, sodass unsere clientseitigen Skripte so kurz und effizient wie möglich werden konnten. Jetzt wollen wir ein anderes Szenarium betrachten, in dem vollständig neue Informationen dargestellt werden, wenn JavaScript verfügbar ist. In zunehmendem Maße greifen ausgereifte Webanwendungen auf JavaScript zurück, um Inhalte anzubieten und sie bei Anlieferung zu manipulieren. In unserem dritten Sortierbeispiel werden wir das ebenfalls tun.

### Hinweis zur allmählichen Funktionsverminderung

Zur Verdeutlichung dieses Beispiels enthält unser HTML-Dokument keinen Inhalt für unsere Tabelle. In einer echten Anwendung würden wir die Tabelle im Voraus mit Inhalten füllen, sodass Benutzer ohne JavaScript zumindest die unsortierten Tabellendaten sehen können.

Wir beginnen mit dem Schreiben zweier Funktionen: `buildRow()`, die den HTML-Code für eine einzelne Tabellenzeile generiert, und `buildRows()`, die mit `$.map()` per Schleife durch alle Zeilen im Datensatz läuft und für jede `buildRow()` aufruft. Auch wenn wir unseren Zweck mit nur einer Funktion erreichen könnten, bieten

zwei getrennte Funktionen die Möglichkeit, zu einem anderen Zeitpunkt auch eine einzelne Zeile zu erzeugen und einzufügen. Die Funktionen bekommen ihre Daten aus der Antwort auf einen Ajax-Aufruf:

```
(function($) {
    function buildRow(row) {
        var authors = [];
        $.each(row.authors, function(index, auth) {
            authors[index] = auth.first_name + ' ' + auth.last_name;
        });

        var html = '<tr>';
        html += '<td></td>';
        html += '<td>' + row.title + '</td>';
        html += '<td>' + authors.join(', ') + '</td>';
        html += '<td>' + row.published + '</td>';
        html += '<td>$' + row.price + '</td>';
        html += '</tr>';

        return html;
    }

    function buildRows(rows) {
        var allRows = $.map(rows, buildRow);
        return allRows.join('');
    }

    $.getJSON('books.json', function(json) {
        $(document).ready(function() {
            var $table3 = $('#t-3');
            $table3.find('tbody').html(buildRows(json));
        });
    });
})(jQuery);
```

**Listing 12–9**

Einige Punkte in diesem Code sind bemerkenswert. Beachten Sie zunächst, dass alle Funktionen außerhalb von `$(document).ready()` definiert werden. Indem wir auf die Callback-Funktion von `$.getJSON()` warten, um `$(document).ready()` aufzurufen, geben wir dem Teil unseres Codes, der nicht vom DOM abhängt, einen Vorsprung.

Beachtenswert ist auch, dass wir die `authors`-Daten anders behandeln müssen, weil sie vom Server als Objekt-Array mit `first_name`- und `last_name`-Eigenschaften geliefert werden, während alles andere als String oder Zahl ankommt. Wir laufen per Schleife durch das `authors`-Array, auch wenn die meisten Arrays nur aus einem Element bestehen, und verbinden Vor- und Nachnamen. Nach der letzten `$.each()`-Schleife führen wir die Array-Werte getrennt durch Komma und Leerzeichen zusammen und erhalten eine schön formatierte Namensliste.

Die Funktion `buildRow()` geht davon aus, dass der in der JSON-Datei enthaltene Text einwandfrei weiterverarbeitet werden kann. Da wir die `<img>`-, `<td>`- und `<tr>`-Tags zusammen mit dem Text in einem einzelnen Textstring zusammenfassen, müssen wir sicherstellen, dass darin kein `<`, `>` oder `&` enthalten ist. Ein Weg, sichere HTML-Strings zu erhalten, besteht darin, sie auf dem Server zu bearbeiten und dabei alle Vorkommen von `<` in `&lt;` usw. zu konvertieren.

Auch wenn wir unsere Tabellenzeilen mit diesen beiden Funktionen liebevoll per Hand zusammengebaut haben, helfen uns JavaScript-Templatesysteme wie Mustache (<https://github.com/janl/mustache.js>) und Handlebars (<http://handlebars.strobeapp.com/>) dabei, einen großen Teil der Stringverarbeitung und -verkettung zu erledigen. Der Einsatz von Templates kann besonders dann sinnvoll sein, wenn ein Projekt in Größe und Komplexität wächst.

### 12.5.1 Das JSON-Objekt modifizieren

Die Bearbeitung des `author`-Arrays ist sinnvoll, wenn wir die `buildRows()`-Funktion nur einmal aufrufen wollen. Da wir sie jedoch immer einsetzen, wenn die Zeilen sortiert werden, sollten wir die Informationen vorher formatieren. Dabei können wir die Titel- und Autorinformationen ebenfalls für die Sortierung aufarbeiten. Anders als in der zweiten Tabelle, in der wir in jeder Zeile sortierbare Daten im `data-book`-Attribut und den Anzeigedaten in den Tabellenzellen hatten, sind die JSON-Daten für die dritte Tabelle einheitlich. Durch nur eine weitere Funktion erreichen wir, dass wir auch modifizierte Werte zum Sortieren und Darstellen verwenden können, bevor wir zu den Funktionen zur Tabellenerstellung kommen:

```
function prepRows(rows) {
    $.each(rows, function(i, row) {
        var authors = [],
            authorsFormatted = [];

        rows[i].titleFormatted = row.title;
        rows[i].title = row.title.toUpperCase();

        $.each(row.authors, function(j, auth) {
            authors[j] = auth.last_name + ' ' + auth.first_name;
            authorsFormatted[j] = auth.first_name +
                ' ' + auth.last_name;
        });
        rows[i].authorsFormatted = authorsFormatted.join(', ');
        rows[i].authors = authors.join(' ').toUpperCase();
    });

    return rows;
}
```

**Listing 12-10**

Indem wir unsere JSON-Daten mit dieser Funktion übergeben, fügen wir jedem Zeilenobjekt zwei Eigenschaften hinzu: `authorsFormatted` und `titleFormatted`. Diese Eigenschaften werden zur Darstellung des Tabelleninhalts verwendet, sodass die ursprünglichen Eigenschaften `authors` und `title` für die Sortierung erhalten bleiben. Die Eigenschaften zur Sortierung werden außerdem in Großbuchstaben konvertiert, um die Sortierung schreibweisenunabhängig zu machen.

Wenn wir diese `prepRows()`-Funktion sofort innerhalb der `$.getJSON()`-Callback-Funktion aufrufen, legen wir den zurückgegebenen Wert des JSON-Objekts in der Variablen `rows` ab und verwenden sie zur Sortierung und zur Erstellung. Das bedeutet, dass wir auch die `buildRow()`-Funktion ändern müssen, um die Vereinfachung auszunutzen, die unsere Vorbereitungsfunktion bietet:

```
function buildRow(row) {
    var html = '<tr>';
    html += '<td></td>';
    html += '<td>' + row.titleFormatted + '</td>';
    html += '<td>' + row.authorsFormatted + '</td>';
    html += '<td>' + row.published + '</td>';
    html += '<td>$' + row.price + '</td>';
    html += '</tr>';

    return html;
}

$.getJSON('books.json', function(json) {
    $(document).ready(function() {
        var $table3 = $('#t-3');
        var rows = prepRows(json);
        $table3.find('tbody').html(buildRows(rows));
    });
});
```

*Listing 12-11*

## 12.5.2 Inhalte bei Bedarf wiederherstellen

Nachdem wir jetzt den Inhalt für die Sortierung und zur Darstellung aufbereitet haben, können wir die Funktion zum Verändern der Spaltentitel und die Sortierroutine implementieren:

```
$.getJSON('books.json', function(json) {
    $(document).ready(function() {
        var $table3 = $('#t-3');
        var rows = prepRows(json);
        $table3.find('tbody').html(buildRows(rows));

        var $headers = $table3.find('thead th').slice(1);
        $headers
            .wrapInner('<a href="#"></a>')
            .addClass('sort');
```

```
$headers.bind('click', function(event) {
    event.preventDefault();
    var $header = $(this),
        sortKey = $header.data('sort').key,
        sortDirection = 1;

    if ($header.hasClass('sorted-asc')) {
        sortDirection = -1;
    }

    rows.sort(function(a, b) {
        var keyA = a[sortKey];
        var keyB = b[sortKey];

        if (keyA < keyB) return -sortDirection;
        if (keyA > keyB) return sortDirection;
        return 0;
    });

    $headers.removeClass('sorted-asc sorted-desc');
    $header.addClass(sortDirection == 1 ? 'sorted-asc' :
        'sorted-desc');

    $table3.children('tbody').html(buildRows(rows));
});
});
});
});
```

**Listing 12-12**

Der Code innerhalb des click-Handlers ist nahezu identisch mit dem Handler für die zweite Tabelle in Listing 12-8. Der einzige sichtbare Unterschied besteht darin, dass wir hier nur einmal je Sortierung Inhalte ins DOM einfügen. In den Tabellen eins und zwei haben wir selbst nach der Optimierung die eigentlichen DOM-Elemente sortiert und sind sie dann einzeln per Schleife durchlaufen, um sie in die neue Reihenfolge zu bringen. In Listing 12-8 werden Tabellenzeilen beispielsweise in einer Schleife wie dieser hier wieder eingefügt:

```
$.each(rows, function(index, row) {
    $table2.children('tbody').append(row);
});
```

Dieses wiederholte Einfügen ins DOM kann aus Sicht der Performance Nachteile aufweisen, besonders bei einer großen Anzahl Zeilen. Vergleichen Sie das mit unserem Ansatz aus Listing 12-12:

```
$table3.children('tbody').html(buildRows(rows));
```

Die Funktion buildRows() gibt einen HTML-String zurück, der die Reihen darstellt und sie in einem Rutsch einfügt, statt die bestehenden Zeilen zu verschieben.

### Messen der Performance

Wir bekommen einen Eindruck von der unterschiedlichen Performance der für die drei Tabellen eingesetzten Techniken, indem wir uns die Tests unter <http://jsperf.com/> *sort-and-insert/2* ansehen. Seien Sie jedoch nicht überrascht, wenn der Code dort nicht exakt mit unserem hier übereinstimmt. Code wie den unseren zu testen ist nicht ganz leicht, weil er mehrere Aufgaben erfüllt. Die genauesten Tests versuchen, den Code auf ein Minimum zu reduzieren, sodass jeder Textlauf nur die relevanten Aktionen misst.

## 12.6 Attributmanipulation für Fortgeschrittene

Inzwischen sind wir beim Abrufen und Setzen von Werten, die mit DOM-Elementen verknüpft sind, recht erfahren. Wir haben dies mit einfachen Methoden wie `.attr()`, `.prop()` und `.css()` erledigt, mit bequemen Abkürzungen wie `.addClass()`, `.css()` und `.val()` und komplexen Aktionen wie bei `.animate()`. Selbst die einfachen Methoden erledigen hinter den Kulissen einiges an Arbeit für uns. Wir können sie noch geschickter für unsere Zwecke einsetzen, wenn wir besser verstehen, was sie genau machen.

### 12.6.1 Elementerstellung per Kurzschrift

Oft erzeugen wir in unserem jQuery-Code neue Elemente, indem wir der `$()`-Funktion oder DOM-Einfügefunktionen einen HTML-String übergeben. Zum Beispiel erstellen wir in Listing 12–9 ein ziemlich großes HTML-Fragment. Dieses Verfahren ist schnell und genau. Es gibt jedoch Umstände, unter denen es nicht ideal ist. Wir könnten beispielsweise Sonderzeichen im Text umwandeln wollen, bevor er verwendet wird, oder browserabhängige Formatregeln anwenden. In diesen Fällen würden wir erst das Element erstellen und dann mit entsprechenden jQuery-Methoden zur Modifikation verketten. Die `$()`-Funktion selbst bietet uns eine durchaus ansprechende Alternativsyntax.

Nehmen wir an, wir wollten Überschriften vor den Tabellen in unserem Dokument einfügen. Wir können `.each()` dazu verwenden, um in einer Schleife über die Tabellen zu laufen und für jede eine passend benannte Überschrift zu erzeugen:

```
$(document).ready(function() {
    $('table').each(function(index) {
        var $table = $(this);
        $('

### </h3>', { id: 'table-title-' + index, 'class': 'table-title', text: 'Table ' + (index + 1), data: {'index': index}, click: function() {


```

```
$table.fadeToggle();
return false;
},
css: {glowColor: '#00ff00'}
}).insertBefore($table);
});
});
});
```

**Listing 12-13**

Das Übergeben einer Map mit Optionen als zweites Argument an die `$(())`-Funktion hat denselben Effekt, wie zuerst das Element zu erzeugen und diese Map an die `.attr()`-Methode zu übergeben. Wie wir wissen, können wir mit dieser Methode DOM-Attribute setzen, beispielsweise die `id` des Elements.

Der Rest der Optionen in unserem Beispiel mag zuerst ein wenig verwirrend aussehen. Wir legen den Text für die Überschriften fest sowie Zusatzdaten, wie beispielsweise einen `click`-Handler. Es handelt sich dabei nicht um DOM-Attribute, auch wenn sie das Gleiche erreichen. Diese verkürzte `$(())`-Syntax kann zusammen mit der `.attr()`-Funktion eine große Anzahl zusätzlicher DOM-Funktionen steuern, indem sie *Hooks* einsetzt.

### 12.6.2 DOM-Manipulation mit Hooks

Viele jQuery-Methoden zum Lesen und Setzen von Eigenschaften können mit passenden Hooks an Sonderfälle angepasst werden. Diese Hooks sind im jQuery-Namensraum Arrays mit Namen wie `$.cssHooks`, `$.attrHooks` und `$.attrFn`. Normalerweise sind Hooks Objekte, die eine `get`-Methode enthalten, die einen angeforderten Wert abruft, und eine `set`-Methode, die einen neuen Wert einrichtet.

Die Map `$.attrFn` bildet eine Ausnahme. Sie dient als Beispiel in Listing 12-13 und enthält einfach eine Liste von `.attr()`-Schlüsseln, die ihre Werte an die entsprechenden jQuery-Werte senden. Da z.B. `text` in `$.attrFn` gelistet ist, wird der String, den wir als Wert für den Schlüssel angeben, an die Methode `.text()` übergeben.

Die anderen Hook-Typen sind:

- `$.attrHooks`, das `.attr()` verändert. Das verhindert beispielsweise, dass das `type`-Attribut eines Elements verändert wird.
- `$.cssHooks`, das `.css()` verändert. Dies dient z.B. zur Steuerung von `opacity` im Internet Explorer.
- `$.propHooks`, das `.prop()` verändert. Zum Beispiel ändert sich damit das Verhalten der Eigenschaft `selected` in Safari.
- `$.valHooks`, das `.val()` verändert. Damit können z.B. Schaltflächen und Markierungsfelder in unterschiedlichen Browsern denselben Wert annehmen.

Normalerweise sind wir von der Arbeitsweise der Hooks vollständig abgeschirmt und wir können ihre Vorteile nutzen, ohne viel über sie nachdenken zu müssen. Manchmal möchten wir das Verhalten einer jQuery-Methode jedoch mit eigenen Hooks verändern.

### Einen CSS-Hook schreiben

Der Code in Listing 12–13 führt auf der Seite eine CSS-Eigenschaft namens `glowColor` ein. Zurzeit hat das keine Auswirkung auf die Seite, da eine solche Eigenschaft nicht existiert. Wir möchten aber `$.cssHooks` dahingehend erweitern, dass es diese neu eingeführte Eigenschaft unterstützt. Dazu fügen wir um den Text ein leichtes Schimmern ein, indem wir die CSS3-Eigenschaft `text-shadow` verwenden, wenn `glowColor` für ein Element gesetzt ist. Da `text-shadow` im Internet Explorer nicht unterstützt wird, werden wir den Schimmer mit der Microsoft-eigenen Eigenschaft `filter` implementieren:

```
(function($) {
    var div = document.createElement('div');
    $.support.textShadow = div.style.textShadow === '';
    $.support.filter = div.style.filter === '';
    div = null;

    if ($.support.textShadow) {
        $.cssHooks.glowColor = {
            set: function(elem, value) {
                if (value == 'none') {
                    elem.style.textShadow = '';
                }
                else {
                    elem.style.textShadow = '0 0 2px ' + value;
                }
            }
        };
    }
    else { $.cssHooks.glowColor = {
        set: function(elem, value) {
            if (value == 'none') {
                elem.style.filter = '';
            }
            else {
                elem.style.zoom = 1;
                elem.style.filter =
                    'progid:DXImageTransform.Microsoft' +
                    '.Glow(Strength=2, Color=' + value + ')';
            }
        }
    };
}
)) (jQuery);
```

**Listing 12–14**

Ein Hook besteht aus einer get- und einer set-Methode für ein Element. Um unser Beispiel so kurz und einfach wie möglich zu halten, definieren wir zu diesem Zeitpunkt nur set. Vor der Definition eines Hooks lassen wir einige Kompatibilitätstests für die Funktion laufen. Wenn text-shadow vom Browser unterstützt wird, definieren wir eine Version des Hooks, die diese Eigenschaft unterstützt. Wenn nicht, prüfen wir auf die Unterstützung durch DirectX-Filter und verwenden diese. Gibt es beides nicht, wird der Hook gar nicht definiert und glowColor hat keine Wirkung.

Mit diesem Hook bekommen wir nun einen grünen Schimmer von zwei Pixeln Breite um die Überschrift herum:

**Table 1**

Obwohl der neue Hook wie gewünscht funktioniert, fehlen ihm viele Funktionen, die wir vielleicht erwarten. Zu diesen Mängeln zählen z.B.:

- Die Größe des Schimmers ist nicht anpassbar.
- Der Effekt schließt die gemeinsame Verwendung mit anderen text-shadow- oder filter-Effekten aus.
- Die get-Callbacks sind nicht implementiert, sodass wir den aktuellen Wert der Eigenschaft nicht prüfen können.
- Die Eigenschaft kann nicht animiert werden.

Mit genügend Arbeit und zusätzlichem Code können wir diese Hindernisse überwinden. In der Praxis müssen wir aber nicht oft eigene Hooks definieren. Fähige Plug-in-Entwickler haben nämlich Hooks für die unterschiedlichsten Bedürfnisse erstellt, darunter auch für die meisten CSS3-Eigenschaften.

#### Wie Sie Hooks finden

Die angebotenen Plug-ins variieren laufend. Es entstehen ständig neue Hooks, sodass es nicht möglich ist, sie alle aufzulisten. Ein Beispiel für das, was machbar ist, finden Sie in Brandon Aarons Sammlung von CSS-Hooks: <https://github.com/brandon-aaron/jquery-cssHooks>.

## 12.7 Zusammenfassung

In diesem Kapitel haben wir ein häufiges Problem – Tabellendaten zu sortieren – auf drei verschiedene Arten gelöst und dabei die Vorteile jedes Ansatzes besprochen. Dadurch haben wir die Techniken zur DOM-Modifikation geübt, die wir früher gelernt haben, und die Methode .data() zum Setzen und Abrufen von Daten untersucht, die mit DOM-Elementen oder über HTML5-Datenattribute verbunden sind. Wir haben auch gezeigt, wie einige DOM-Modifikationsroutinen funktionieren, und gelernt, wie wir sie für unsere Zwecke erweitern.

### **12.7.1 Literatur**

Eine vollständige Liste der DOM-Manipulationsmethoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## **12.8 Übungsaufgaben**

Um die folgenden Übungsaufgaben durchführen zu können, benötigen Sie die Datei index.html für dieses Kapitel sowie den fertigen JavaScript-Code aus complete.js. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Verändern Sie die Schlüsselberechnung für die erste Tabelle so, dass Titel und Autor nach ihrer Länge sortiert werden und nicht alphabetisch.
2. Verwenden Sie die HTML5-Daten in der zweiten Tabelle, um die Summe aller Buchpreise zu bilden, und fügen Sie sie in die Überschrift der Spalte ein.
3. Ändern Sie den Komparator für die dritte Tabelle so, dass Titel, die das Wort jQuery enthalten, in der Sortierung vorn auftauchen.
4. *Schwierig:* Implementieren Sie get-Callbacks für den CSS-Hook glowColor.



# 13 Ajax für Fortgeschrittene

Viele Webanwendungen benötigen häufige Kommunikation mit dem Netzwerk. In Kapitel 6 haben wir Wege untersucht, wie wir Informationen mit dem Server austauschen, ohne im Browser neue Seiten laden zu müssen. Diese Ajax-Techniken sind sehr nützlich und zählen zu den anspruchsvollsten Dingen, die wir mit jQuery anstellen können.

In diesem Kapitel steigen wir tiefer in die Fähigkeiten des Ajax-Frameworks von jQuery ein. Wir betrachten gute Ansätze zur Fehlerbehandlung bei Netzwerkstörungen, untersuchen das Zusammenspiel von Ajax und dem jQuery-System für verzögerte Objekte und entwickeln Wege, um unsere Netzwerkübertragungen durch Caching und Drosselung auf ein Minimum zu beschränken. Schließlich sehen wir uns die interne Funktionsweise des Ajax-Systems an, sodass wir seine Funktionen bei Bedarf erweitern können.

## 13.1 Fortschreitende Verbesserung mit Ajax

Wir haben das Konzept der fortschreitenden Verbesserung bereits an vielen Stellen angetroffen. Dieser Ansatz bietet jedem Benutzer einen hohen Nutzwert, indem zuerst auf eine funktionsfähige Anwendung geachtet wird, bevor die Erweiterungen für moderne Browser dazukommen.

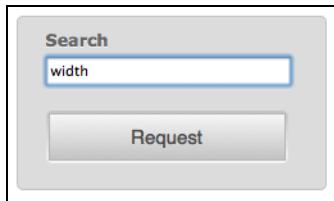
Ajax-lastige Anwendungen laufen Gefahr, nicht mehr zu funktionieren, wenn JavaScript deaktiviert ist. Um das zu verhindern, können wir zuerst eine klassische Client-Server-Seite mit Formularen erstellen und diese Formulare leistungsfähiger machen, wenn JavaScript als Unterstützung zur Verfügung steht.

Als Beispiel erstellen wir ein Formular, das die jQuery-API-Dokumentation durchsucht. Da es zu diesem Zweck bereits ein Formular auf der jQuery-Seite gibt, können wir unser eigenes Formular darauf aufsetzen:

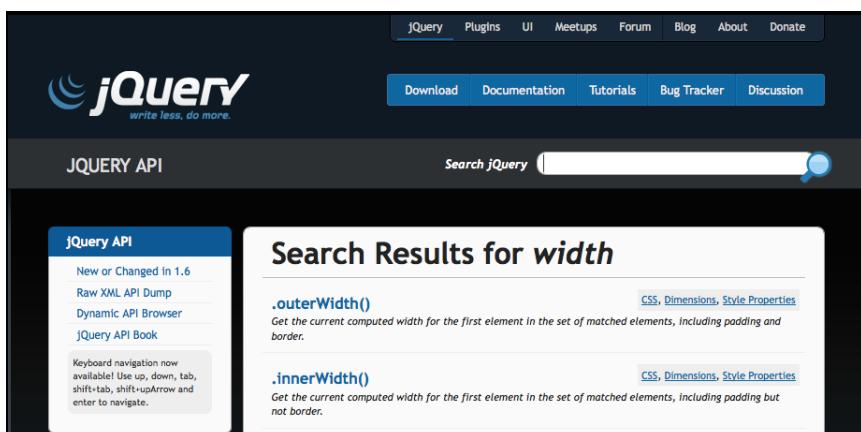
```
<form id="ajax-form" action="http://api.jquery.com/" method="get">
  <fieldset>
    <div class="text">
      <label for="title">Search</label>
      <input type="text" id="title" name="s">
    </div>
```

```
<div class="actions">
    <button type="submit">Request</button>
</div>
</fieldset>
</form>
```

Das Suchformular besitzt einige Stilelemente aus unserer CSS-Datei, ist aber ansonsten ein normales Element mit Text eingabefeld und einer Schaltfläche zum Senden.



Wenn die REQUEST-Schaltfläche im Formular angeklickt wird, dann wird das Formular normal übertragen, der Benutzer wird auf <http://api.jquery.com/> geleitet und die Ergebnisse werden wie folgt angezeigt:

A screenshot of the jQuery API search results for the term "width". The search bar at the top shows "Search jQuery" with a magnifying glass icon. On the left, there's a sidebar with links like "jQuery API", "New or Changed in 1.6", "Raw XML API Dump", "Dynamic API Browser", and "jQuery API Book". The main content area is titled "Search Results for width" and lists two results: ".outerWidth()" and ".innerWidth()". Both results are categorized under "CSS, Dimensions, Style Properties". The description for ".outerWidth()" says: "Get the current computed width for the first element in the set of matched elements, including padding and border." The description for ".innerWidth()" says: "Get the current computed width for the first element in the set of matched elements, including padding but not border."

Ist JavaScript verfügbar, wollen wir diesen Inhalt in einen #response-Container unserer Suchseite laden, statt sie zu verlassen. Wären die Daten auf demselben Server wie unser Suchformular, könnten wir den relevanten Teil mit der .load()-Methode einsammeln:

```
$(document).ready(function() {
    var $ajaxForm = $('#ajax-form'),
        $response = $('#response');
```

```
$ajaxForm.bind('submit', function(event) {  
    event.preventDefault();  
    $response.load('http://api.jquery.com/ #content',  
        $ajaxForm.serialize());  
});  
});
```

**Listing 13-1**

Da sich die API-Seite jedoch unter einem anderen Hostnamen verbirgt, erlaubt die Cross-Domain-Richtlinie des Browsers diese Transaktion aber nicht. Stattdessen müssen wir eine Methode einsetzen, die domainübergreifend funktioniert. Glücklicherweise bietet die API-Seite die Daten im JSONP-Format an, das für unsere Zwecke perfekt geeignet ist.

#### Cross-Domain-Abfragen mit Yahoo! Query Language

Einige Drittanbieterseiten stellen jedoch keine JSONP-Such-API zur Verfügung. Wenn wir auf Daten solcher Seiten zugreifen müssen und ein eigener serverseitiger Proxy nicht möglich ist, kann die Yahoo! Query Language (YQL) diese Lücke schließen. Sie setzt ihren eigenen Proxy auf, sodass wir die Abfrage mit den entsprechenden Parametern an den YQL-Service senden können und als Antwort JSONP erhalten. Weitere Informationen über YQL erhalten Sie unter <http://developer.yahoo.com/yql/> und einen Blog mit Anweisungen und Beispielen für die Verwendung von YQL mit jQuery finden Sie unter <http://www.wait-till-i.com/2010/01/10/loading-external-content-with-ajax-using-jquery-and-yql/>.

### 13.1.1 JSONP-Daten einsammeln

In Kapitel 6 haben wir gesehen, dass JSONP einfach JSON mit einer zusätzlichen Schicht mit Serverfunktionen ist, die Anfragen von einer anderen Seite aus zulässt. Kommt eine Anfrage nach JSONP-Daten, wird ein spezieller Anfrage-Stringparameter bereitgestellt, der es dem anfragenden Skript ermöglicht, die Daten einzusammeln. Dieser Parameter kann genannt werden, wie es der JSONP-Server wünscht. Im Falle unserer jQuery-API-Seite wird der Parameter `callback` genannt, was die Voreinstellung ist.

Da der vorgegebene Callback-Name verwendet wird, besteht der einzige Schritt für eine JSONP-Abfrage darin, jQuery mitzuteilen, dass `jsonp` der erwartete Datentyp ist:

```
$(document).ready(function() {  
    var $ajaxForm = $('#ajax-form'),  
        $response = $('#response');  
  
    $ajaxForm.bind('submit', function(event) {  
        event.preventDefault();  
    });  
});
```

```
$.ajax({
    url: 'http://api.jquery.com/jsonp/',
    dataType: 'jsonp',
    data: {
        title: $('#title').val()
    },
    success: function(data) {
        console.log(data);
    }
});
});
```

**Listing 13–2**

Jetzt können wir die JSON-Daten in der Konsole untersuchen. In diesem Fall sind die Daten ein Array von Objekten, die jeweils eine jQuery-Methode beschreiben:

```
{
    "url": "http://api.jquery.com/innerWidth/",
    "name": "innerWidth",
    "title": ".innerWidth()",
    "type": "method",
    "signatures": [
        {
            "added": "1.2.6"
        }
    ],
    "desc": "Get the current computed width for the first element in
the set of matched elements, including padding but not
border.",
    "longdesc": "<p>This method returns the width of the element,
including left and right padding, in pixels.</p>\n<p>This
method is not applicable to <code>window</code> and
<code>document</code> objects; for these, use <code><a
href=\"/width\">.width()</a></code> instead.</p>\n<p
class=\"image\"><img src=\"/images/0042_04_05.png\"/></p>",
    "categories": [
        "CSS",
        "Dimensions",
        "Manipulation > Style Properties",
        "Version > Version 1.2.6"
    ],
    "download": ""
}
```

Alle Daten, die wir über eine Methode anzeigen müssen, sind in diesem Objekt enthalten. Wir müssen sie nur anzeigetauglich formatieren. Den HTML-Code für ein Element zu erstellen kann aufwendig sein, sodass wir diesen Schritt übergehen und in seine eigene Hilfsfunktion auslagern:

```
var buildItem = function(item) {
    var title = item.name,
        args = [],
        output = '<li>';

    if (item.type == 'method' || !item.type) {
        if (item.signatures[0].params) {
            $.each(item.signatures[0].params, function(index, val) {
                args.push(val.name);
            });
        }
        title = (/^jQuery|deferred/).test(title)
            ? title : '.' + title;
        title += '(' + args.join(', ') + ')';
    } else if (item.type == 'selector') {
        title += ' selector';
    }
    output += '<h3><a href="' + item.url + '">' +
        title + '</a></h3>';
    output += '<div>' + item.desc + '</div>';
    output += '</li>';

    return output;
};


```

**Listing 13–3**

Die Funktion `buildItem()` wandelt das JSON-Objekt in ein HTML-Listenelement um. Da wir berücksichtigen müssen, auch mehrere Methodenargumente oder Funktionssignaturen bearbeiten zu können, setzen wir Schleifen und `.join()`-Aufrufe dazu ein. Ist das erledigt, erzeugen wir einen Link auf die Hauptdokumentation und geben die Elementbeschreibung aus.

Jetzt verfügen wir über eine Funktion, die den HTML-Code für ein einzelnes Element erzeugt. Wenn unser Ajax-Aufruf abgeschlossen ist, müssen wir die Funktion für jedes zurückgegebene Objekt aufrufen und alle Ergebnisse anzeigen:

```
$(document).ready(function() {
    var $ajaxForm = $('#ajax-form'),
        $response = $('#response'),
        noresults = 'There were no search results.';

    var response = function(json) {
        var output = '';
        if (json && json.length) {
            output += '<ol>';
            $.each(json, function(index, val) {
                output += buildItem(val);
            });
            output += '</ol>';
        }
    };
});
```

```
        } else {
            output += noresults;
        }

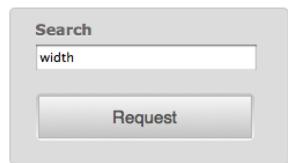
        $response.html(output);
    };
$ajaxForm.bind('submit', function(event) {
    event.preventDefault();

    $.ajax({
        url: 'http://api.jquery.com/jsonp/',
        dataType: 'jsonp',
        data: {
            title: $('#title').val()
        },
        success: response
    });
});
});
```

**Listing 13–4**

Wir haben den Success-Handler aus den `$.ajax()`-Optionen herausgezogen, sodass wir ihn durch seinen Variablenamen ansprechen können. Auch wenn wir anonyme Inline-Funktionen für einen kompakteren Code einsetzen, hindert uns nichts daran, Funktionen zu separieren und zu benennen, um den Code besser lesbar zu gestalten.

Jetzt haben wir einen funktionsfähigen Success-Handler, der die Suchergebnisse ansprechend in einer Spalte neben unserem Formular anzeigt.



The screenshot shows a simple web interface. At the top is a header with the word "Search". Below it is a search input field containing the placeholder "width". Underneath the input is a button labeled "Request". To the right of the search form is a table with two columns. The first column contains the search term "width". The second column contains the result "1. innerWidth()". Below the table, there is a detailed description of the innerWidth() method.

1. **.innerWidth()**  
Get the current computed width for the first element in the set of matched elements, including padding but not border.

2. **.outerWidth(includeMargin)**  
Get the current computed width for the first element in the set of matched elements, including padding and border.

3. **.width()**  
Get the current computed width for the first element in the set of matched elements.

4. **.width(value)**  
Set the CSS width of each element in the set of matched elements.

## 13.2 Ajax-Fehlerbehandlung

Jede Netzwerkoperation in einer Anwendung bringt ein gewisses Maß an Unsicherheit mit sich. Die Verbindung des Benutzers kann mitten in einer Operation abbrechen oder der Server für eine Weile ausfallen. Wegen dieser Unsicherheiten müssen wir den Worst Case immer in Betracht ziehen und Fehlerszenarien entwickeln.

Die `$.ajax()`-Funktion kann in diesen Fällen eine Callback-Funktion namens `error` aufrufen. In diesem Callback geben wir dem Benutzer eine Art Feedback, um anzusehen, dass ein Fehler aufgetreten ist:

```
$(document).ready(function() {
    var $ajaxForm = $('#ajax-form'),
        $response = $('#response'),
        noresults = 'There were no search results.',
        failed = 'Sorry, but the request could not ' +
            'reach its destination. Try again later.';

    $ajaxForm.bind('submit', function(event) {
        event.preventDefault();

        $.ajax({
            url: 'http://api.jquery.com/jsonp/',
            dataType: 'jsonp',
            data: {
                title: $('#title').val()
            },
            success: response,
            error: function() {
                $response.html(failed);
            }
        });
    });
});
```

**Listing 13–5**

Der `error`-Callback kann u.a. aus folgenden Gründen aufgerufen werden:

- Der Server hat einen Statuscode wie 403 Forbidden, 404 Not Found oder 500 Internal Server Error zurückgeliefert.
- Der Server hat einen Umleitungscode wie 301 Moved Permanently gesendet. Eine Ausnahme bildet 304 Not Modified, das keinen Fehler auslöst, weil der Browser damit umgehen kann.
- Die vom Server zurückgegebenen Daten können nicht wie angegeben bearbeitet werden (weil sie z.B. kein gültiges JSON sind, obwohl der Datentyp json ist).
- Die `.abort()`-Methode wird von Objekt XMLHttpRequest aufgerufen.

Diese Zustände zu erkennen und darauf zu reagieren ist wichtig für den Bedienkomfort des Benutzers. Wir haben in Kapitel 6 gesehen, dass uns der Fehlercode, falls vorhanden, über die `.status`-Eigenschaft des jqXHR-Objekts geliefert wird, das an den `error`-Callback übergeben werden kann. Wir können den Wert von `jqXHR.status` auch einsetzen, um auf unterschiedliche Fehlerarten entsprechend zu reagieren.

Die Fehlermeldungen des Servers sind jedoch nur dann sinnvoll, wenn wir sie auch im Auge behalten. Einige Fehler werden sofort erkannt, aber andere Zustände können eine längere Verzögerung zwischen Anfrage und möglicher Fehlermeldung verursachen.

Wenn kein zuverlässiger Servertimeout-Mechanismus zur Verfügung steht, können wir ein eigenes clientseitiges Anfrage-Timeout einsetzen. Indem wir der Timeout-Option eine Dauer in Millisekunden angeben, teilen wir `$.ajax()` mit, nach diesem Zeitraum `.abort()` aufzurufen, wenn keine Antwort eintrifft:

```
$ajax({
  url: 'http://api.jquery.com/jsonp/',
  dataType: 'jsonp',
  data: {
    title: $('#title').val()
  },
  timeout: 15000,
  success: response,
  error: function() {
    $response.html(failed);
  }
});
```

**Listing 13–6**

Ist das Timeout gesetzt, können wir davon ausgehen, dass die Daten binnen 15 Sekunden bereitstehen oder der Benutzer eine Fehlermeldung erhält.

### 13.3 Das jqXHR-Objekt

Wenn ein Ajax-Aufruf erfolgt, legt jQuery den besten Weg zum Abruf der Daten fest. Dieser Transport kann über das XMLHttpRequest-Objekt erfolgen, das Microsoft-ActiveX-XMLHTTP-Objekt oder über ein `<script>`-Tag.

Da die eingesetzte Transportart von Aufruf zu Aufruf variieren kann, benötigen wir eine Standardschnittstelle für die Kommunikation. Das Objekt jqXHR bietet uns diese Schnittstelle: Es verpackt das XMLHttpRequest-Objekt, wenn diese Transportart eingesetzt wird und in anderen Fällen simuliert es XMLHttpRequest so gut es geht. Zu den genutzten Eigenschaften und Methoden gehören:

- `.responseText` oder `.responseXML`, in dem die zurückgegebenen Daten enthalten sind.
- `.status` und `.statusText`, in dem der Statuscode und die Beschreibung enthalten sind.
- `.setRequestHeader()`, um die mit der Anfrage gesendeten HTTP-Header zu verändern.
- `.abort()`, um die Transaktion vorzeitig zu stoppen.

Dieses jqXHR-Objekt wird von allen Ajax-Methoden von jQuery zurückgegeben, sodass wir das Ergebnis speichern können, wenn wir Zugriff auf die Eigenschaften oder Methoden benötigen.

### 13.3.1 Ajax-Promises

Ein vielleicht wichtigerer Aspekt von jqXHR als die XMLHttpRequest-Schnittstelle ist seine Funktion als *Promise*. In Kapitel 11 haben wir verzögerte Objekte kennengelernt, mit denen wir Callbacks beim Abschluss bestimmter Operationen auslösen konnten. Ajax-Aufrufe sind eine solche Operation und das jqXHR-Objekt bietet uns die Methoden, die wir von einem Promise eines verzögerten Objekts erwarten.

Mit den Methoden des Promise können wir unseren `$.ajax()`-Aufruf umschreiben, um die Callbacks `success` und `error` durch eine neue Syntax zu ersetzen:

```
$.ajax({
  url: 'http://api.jquery.com/jsonp/',
  dataType: 'jsonp',
  data: {
    title: $('#title').val()
  },
  timeout: 15000
})
.done(response)
.fail(function() {
  $response.html(failed);
});
```

**Listing 13-7**

Auf den ersten Blick erscheint es nicht besser zu sein, `.done()` und `.fail()` aufzurufen. Die Promise-Methoden bieten jedoch einige Vorteile. Zum einen können diese Methoden mehrfach aufgerufen werden, um mehrere Handler zu erhalten. Wenn wir das Ergebnis des `$.ajax()`-Aufrufs in einer Variablen speichern, können wir die Handler später wieder anfügen, was unsere Codestruktur besser lesbar macht. Außerdem werden die Handler sofort aufgerufen, wenn die Ajax-Operation bereits abgeschlossen ist. Und zuletzt sollten wir nicht den Vorteil der besseren Lesbarkeit vergessen, den wir durch Verwenden einer Syntax erhalten, die mit anderen Bestandteilen der jQuery-Bibliothek konsistent ist.

Als weiteres Beispiel für den Einsatz der Promise-Methoden können wir bei Anfragen einen Fortschrittsbalken einblenden. Da wir den Balken ausblenden möchten, wenn der Ladevorgang abgeschlossen ist, egal ob erfolgreich oder auch nicht, setzen wir die `.always()`-Methode ein:

```
$ajaxForm.bind('submit', function(event){  
    event.preventDefault();  
  
    $response.addClass('loading').empty();  
  
    $.ajax({  
        url: 'http://api.jquery.com/jsonp/',  
        dataType: 'jsonp',  
        data: {  
            title: $('#title').val()  
        },  
        timeout: 15000  
    })  
    .done(response)  
    .fail(function() {  
        $response.html(failed);  
    })  
    .always(function() {  
        $response.removeClass('loading');  
    });  
});
```

**Listing 13-8**

Bevor wir einen `$.ajax()`-Aufruf verwenden, fügen wir dem Antwortcontainer die Klasse `loading` hinzu. Ist der Ladevorgang abgeschlossen, entfernen wir sie wieder. Auf diese Weise haben wir den Benutzerkomfort erneut erweitert.

Um einen richtigen Eindruck von den Möglichkeiten der Promises zu bekommen, sehen wir uns im Folgenden an, was wir machen können, wenn wir das Ergebnis unseres `$.ajax()`-Aufrufs in einer Variablen ablegen.

### 13.3.2 Antworten cachen

Wenn wir bestimmte Daten wiederholt benötigen, sollten wir nicht jedes Mal einen Ajax-Aufruf starten, sondern die zurückgegebenen Daten in einer Variablen speichern. Wenn wir diese Daten verwenden müssen, prüfen wir, ob sie sich bereits in der Variablen befinden. Wenn ja, nutzen wir sie. Wenn nicht, müssen wir eine Ajax-Anfrage starten und die Daten in ihrem `.done()`-Handler im Cache ablegen und dann mit ihnen arbeiten.

Das sind viele Schritte. Nutzen wir die Eigenschaften der Promises, kann die Sache aber recht einfach sein, wie im folgenden Codeabschnitt gezeigt:

```
var api = {};  
  
$ajaxForm.bind('submit', function(event) {  
    event.preventDefault();  
  
    $response.empty();
```

```
var search = $('#title').val();
if (search == '') {
    return;
}

$response.addClass('loading');

if (!api[search]) {
    api[search] = $.ajax({
        url: 'http://api.jquery.com/jsonp/',
        dataType: 'jsonp',
        data: {
            title: search
        },
        timeout: 15000
    });
}
api[search].done(response).fail(function() {
    $response.html(failed);
}).always(function() {
    $response.removeClass('loading');
});
});
```

**Listing 13–9**

Wir haben eine neue Variable namens `api` eingeführt, um die erzeugten jqXHR-Objekte festzuhalten. Diese Variable ist ein Objekt mit Schlüsseln, die ausgeführten Suchen entsprechen. Wird das Formular übertragen, prüfen wir, ob es für diesen Schlüssel bereits ein jqXHR-Objekt gibt. Wenn nicht, führen wir die Abfrage durch – wie vorher auch – und speichern das zurückgegebene Objekt in `api`.

Die Handler `.done()`, `.fail()` und `.always()` werden dann an das jqXHR-Objekt angefügt. Dies geschieht ungeachtet eines Ajax-Aufrufs. Dabei müssen wir nur zwei Situationen berücksichtigen.

Zuerst muss die Ajax-Anfrage, falls noch nicht geschehen, gesendet werden. Das entspricht dem vorherigen Verhalten: Die Anfrage wird gesendet und wir verwenden die Promise-Methoden, um dem jqXHR-Objekt Handler anzufügen. Kommt vom Server eine Antwort zurück, werden die entsprechenden Callbacks ausgelöst und das Ergebnis auf dem Bildschirm angezeigt.

Wenn wir die Suche jedoch schon ausgeführt haben, befindet sich das jqXHR-Objekt bereits in `api`. In diesem Fall muss keine neue Suche erfolgen, aber wir rufen die Promise-Methoden des gespeicherten Objekts trotzdem auf. Dadurch werden dem Objekt neue Handler zugewiesen, da das verzögerte Objekt jedoch schon aufgelöst ist, werden die entsprechenden Handler sofort ausgelöst.

Das jQuery-System für verzögerte Objekte erledigt für uns alle notwendigen Aufgaben. Mit ein paar Zeilen Code haben wir das Problem doppelter Netzwerkanfragen für die Anwendung gelöst.

## 13.4 Ajax-Anfragen drosseln

Eine immer beliebtere Funktion ist es, während der Benutzereingaben bereits eine dynamische Liste mit Vorschlägen anzuzeigen. Wir können diese Livesuche mit der jQuery-API simulieren, indem wir einen Handler an das keyup-Ereignis binden:

```
$('#title').bind('keyup', function(event) {  
    ajaxForm.triggerHandler('submit');  
});
```

**Listing 13–10**

Hier lösen wir einfach den submit-Handler des Formulars aus, wenn der Benutzer etwas ins Suchfeld eingibt. Das kann zu vielen Netzanfragen in schneller Folge führen, abhängig von der Tippgeschwindigkeit des Benutzers. Dadurch wird die Performance von JavaScript beeinträchtigt, das Netzwerk belastet und der Server könnte mit dieser Art Anfragen überfordert sein. Wir haben die Menge der Netzwerkanfragen bereits durch das Anfrage-Caching reduziert. Wir können die Serverbelastung jedoch weiter verringern, indem wir die Anfragen drosseln. In Kapitel 10 haben wir das Konzept der Drosselung mit einem throttledScroll-Ereignis erklärt, das die Aufrufe der integrierten scroll-Ereignisse reduzierte. In diesem Fall wollen wir die Aufrufe beim keyup-Ereignis reduzieren:

```
var searchTimeout,  
    searchDelay = 300;  
  
$('#title').bind('keyup', function(event) {  
    clearTimeout(searchTimeout);  
    searchTimeout = setTimeout(function() {  
        ajaxForm.triggerHandler('submit');  
    }, searchDelay);  
});
```

**Listing 13–11**

Unser Verfahren unterscheidet sich ein wenig von dem aus Kapitel 10. Während wir damals den scroll-Handler beim Scrollen mehrfach aufrufen mussten, soll hier das keyup-Verhalten nur einmal nach einem Tastendruck auftreten. Um das zu erreichen, verfolgen wir einen JavaScript-Timer, der anläuft, wenn der Benutzer eine Taste drückt. Jeder Tastendruck setzt diesen Timer zurück, sodass der submit-Handler erst 300 Millisekunden nach dem letzten Tastendruck ausgelöst und die Ajax-Anfrage durchgeführt wird.

## 13.5 Ajax-Funktionen erweitern

Das Ajax-Framework von jQuery ist leistungsfähig, aber wie wir bereits wissen, gibt es Situationen, in denen wir sein Verhalten modifizieren möchten. Es ist keine Überraschung, dass es dazu mehrere Hooks gibt, die Plug-ins für völlig neue Funktionen verwenden können.

### 13.5.1 Konverter für Datentypen

In Kapitel 6 haben wir gesehen, wie die `$.ajaxSetup()`-Funktion es uns ermöglicht, die Voreinstellungen von `$.ajax()` zu verändern und damit viele Ajax-Operationen mit einer Anweisung zu modifizieren. Dieser Mechanismus kann auch eingesetzt werden, um den Bereich der von `$.ajax()` interpretierbaren Datentypen zu erweitern.

Zum Beispiel könnten wir einen Konverter für das Datenformat YAML hinzufügen. YAML (<http://www.yaml.org>) ist ein beliebtes Darstellungsformat, das in vielen Programmiersprachen eingesetzt wird. Falls unser Skript mit einem solchen alternativen Format umgehen können muss, ermöglicht es uns jQuery, die Kompatibilität direkt in die nativen Ajax-Funktionen zu integrieren.

Eine einfache YAML-Datei mit jQuery-Methodenkategorien und Unterkategorien sieht wie folgt aus:

- Ajax:
  - Global Ajax Event Handlers
  - Helper Functions
  - Low-Level Interface
  - Shorthand Methods
  
- Effects:
  - Basics
  - Custom
  - Fading
  - Sliding

Wir können einen bestehenden YAML-Parser, wie den von Diogo Costa (<http://code.google.com/p/javascript-yaml-parser/>), in jQuery verpacken, damit `$.ajax()` diese Sprache ebenfalls versteht.

Einen neuen Ajax-Datentyp zu definieren bedeutet, drei Eigenschaften an `$.ajaxSetup()` zu übergeben: `accepts`, `contents` und `converters`.

Die `accept`-Eigenschaft fügt Header an, die an den Server gesendet werden und die deklarieren, dass von unserem Skript bestimmte MIME-Typen verstanden werden. Die Eigenschaft `contents` behandelt die andere Seite der Transaktion und bietet uns einen regulären Ausdruck, der gegen den MIME-Typ geprüft wird, um den Datentyp automatisch aus den Metadaten zu ermitteln versuchen.

Schließlich enthält converters die eigentlichen Funktionen, die die zurückgegebenen Daten konvertieren:

```
$.ajaxSetup({
    accepts: {
        yaml: 'application/x-yaml, text/yaml'
    },
    contents: {
        yaml: '/yaml/'
    },
    converters: {
        'text yaml': function(textValue) {
            console.log(textValue);
            return '';
        }
    }
});

$.ajax({
    url: 'categories.yml',
    dataType: 'yaml'
});
```

**Listing 13-12**

Die teilweise Implementierung in Listing 13–12 verwendet \$.ajax(), um in der YAML-Datei zu lesen, und deklariert den Datentyp als yaml. Da die Eingangsdaten als text interpretiert werden, benötigt jQuery eine Möglichkeit, einen Datentyp in einen anderen zu konvertieren. Der Schlüssel 'text yaml' von converters zeigt jQuery, dass diese Funktion Daten vom Typ text akzeptiert und sie als yaml uminterpretiert.

In der Konvertierungsfunktion entfernen wir einfach den Textinhalt, um sicherzustellen, dass die Funktion korrekt aufgerufen wird. Um die eigentliche Konvertierung durchzuführen, müssen wir die YAML-Bibliothek des Drittanbieters laden und ihre Methoden wie folgt aufrufen:

```
$.ajaxSetup({
    accepts: {
        yaml: 'application/x-yaml, text/yaml'
    },
    contents: {
        yaml: '/yaml/'
    },
    converters: {
        'text yaml': function(textValue) {
            var result = YAML.eval(textValue);
            var errors = YAML.getErrors();
            if (errors.length) {
                throw errors;
            }
        }
    }
});
```

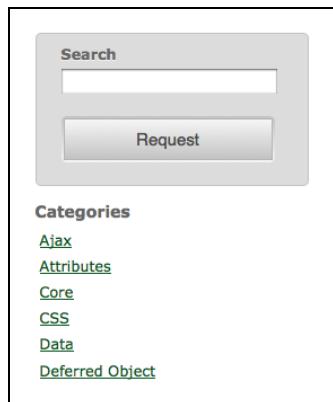
```
        return result;
    }
}
});

$.getScript('yaml.js').done(function() {
$.ajax({
    url: 'categories.yml',
    dataType: 'yaml'
}).done(function (data) {
var cats = '';
$.each(data, function(category, subcategories) {
    cats += '<li><a href="#">' + category + '</a></li>';
});
$(document).ready(function() {
    var $cats = $('#categories').removeClass('hide');
    $('ul', {
        html: cats
    }).appendTo($cats);
});
});
});
});
```

**Listing 13–13**

Die `yaml.js`-Datei enthält ein Objekt namens `YAML` mit den Methoden `.eval()` und `.getErrors()`. Wir verwenden diese Methoden, um den eingehenden Text umzuwandeln, und geben das Ergebnis in Form eines JavaScript-Objekts zurück, das alle Daten der `categories.yml`-Datei in einer einfach zu durchlaufenden Struktur enthält.

Da die Datei, die wir laden, Kategorien von jQuery-Methoden enthält, verwenden wir die konvertierte Struktur, um die Hauptkategorien auszugeben und dem Benutzer später zu ermöglichen, die Suchergebnisse durch Klick auf eine der Kategorien zu filtern:



Wenn wir jetzt Kategorienamen einfügen, müssen wir diesen Teil des Codes in einen `$(document).ready()`-Aufruf verpacken. Die Ajax-Operationen können sofort ausgeführt werden, ohne dass auf das DOM zugegriffen wird, aber wenn wir von ihnen ein Ergebnis erhalten, müssen wir warten, bis das DOM bereit ist, fortzufahren. Indem wir unseren Code so strukturieren, lassen sich Aufgaben frühestmöglich ausführen, sodass die gefühlte Ladezeit der Seite merklich sinkt.

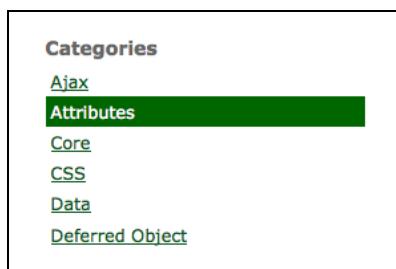
Als Nächstes müssen wir die Klicks auf die Kategorie-Links wie folgt behandeln:

```
$('#categories a').live('click', function(event) {  
    event.preventDefault();  
    $(this).parent().toggleClass('active')  
        .siblings('.active').removeClass('active');  
    $('#ajax-form').triggerHandler('submit');  
});
```

**Listing 13-14**

Indem wir `.live()` an unseren `click`-Handler binden, verhindern wir kostspielige Wiederholungen und wir können den Code sofort ausführen, ohne uns darum zu sorgen, ob der Ajax-Aufruf bereits abgeschlossen ist oder nicht.

Innerhalb des Handlers stellen wir sicher, dass die richtige Kategorie hervorgehoben ist, und lösen dann den `submit`-Handler des Formulars aus. Noch kann das Formular die Kategorieliste nicht interpretieren, aber die Hervorhebung funktioniert bereits:



Schließlich müssen wir den `submit`-Handler des Formulars auf die aktive Kategorie aktualisieren, wenn es eine gibt:

```
$ajaxForm.bind('submit', function(event) {  
    event.preventDefault();  
  
    $response.empty();  
  
    var title = $('#title').val(),  
        category = $('#categories').find('li.active').text(),  
        search = category + '-' + title;  
    if (search == '-') {  
        return;  
    }  
})
```

```
$response.addClass('loading');

if (!api[search]) {
    api[search] = $.ajax({
        url: 'http://api.jquery.com/jsonp/',
        dataType: 'jsonp',
        data: {
            title: title,
            category: category
        },
        timeout: 15000
    });
}
api[search].done(response).fail(function() {
    $response.html(failed);
}).always(function() {
    $response.removeClass('loading');
});
});
```

**Listing 13–15**

Statt den Wert nur aus dem Suchfeld auszulesen, rufen wir jetzt auch den Text aus der aktiven Kategorie ab und geben beide Informationen in einem Ajax-Aufruf weiter. Wir ändern außerdem die Variable search, sodass sie Kategorie und Titel enthalten kann.

Auf diese Weise unterscheidet unser Cache mit Suchergebnissen auch korrekt zwischen Texten aus unterschiedlichen Kategorien.

Jetzt können wir alle Methoden in einer Kategorie anzeigen, indem wir auf einen Kategorienamen klicken oder die Kategorieliste einsetzen, um die Treffer aus der Suche im Eingabefeld einzuschränken.

The screenshot shows a user interface with a search form and a list of categories. On the left, there is a 'Search' form with a text input containing 'stop' and a 'Request' button below it. On the right, under the heading 'Categories', there is a list of items: 'Ajax' (which is highlighted in green), 'Attributes', 'Core', 'CSS', 'Data', and 'Deferred Object'. A tooltip for 'Ajax' provides information about the `.ajaxStop(handler())` method.

**1. `.ajaxStop(handler())`**  
Register a handler to be called when all Ajax requests have completed. This is an [Ajax Event](#).

Zusätzliche Datentypen können auf ähnliche Weise wie in unserem YAML-Beispiel definiert werden. So lässt sich die Ajax-Bibliothek von jQuery an unsere Bedürfnisse anpassen.

### 13.5.2 Ajax-Prefilter

Die `$.ajaxPrefilter()`-Funktion kann *Prefilter* hinzufügen, bei denen es sich um Callback-Funktionen handelt, durch die wir Anfragen verändern können, bevor sie gesendet werden. Prefilter werden aktiviert, bevor `$.ajax()` eine seiner Optionen ändert oder aufruft, sodass sie sich gut dazu eignen, um Optionen zu modifizieren oder neue, eigene Optionen hinzuzufügen.

Prefilter können außerdem den Datentyp einer Anfrage verändern, indem einfach der neue zu verwendende Datentyp zurückgegeben wird. In unserem YAML-Beispiel haben wir `yml` als Datentyp angegeben, weil wir uns nicht darauf verlassen wollten, dass der Server den korrekten MIME-Typ in seiner Antwort verwendet. Wir können natürlich auch einen Prefilter einsetzen, der sicherstellt, dass der Datentyp `yml` ist, wenn sich die entsprechende Dateierweiterung (`.yml`) in der angeforderten URL befindet:

```
$.ajaxPrefilter(function(options) {  
    if (/\.yml$/.test(options.url)) {  
        return 'yml';  
    }  
});
```

**Listing 13-16**

Ein kurzer regulärer Ausdruck prüft, ob sich `.yml` am Ende von `options.url` befindet, und definiert den Datentyp als `yml`, wenn das der Fall ist. Mit diesem Prefilter muss unser Ajax-Aufruf, der das YAML-Dokument holt, seinen Datentyp nicht mehr explizit angeben.

### 13.5.3 Alternative Transporte

Wir haben gesehen, dass jQuery XMLHttpRequest, ActiveX oder `<script>`-Tags einsetzt, um Ajax-Transaktionen entsprechend zu behandeln. Wenn wir möchten, können wir dieses Arsenal mittels neuer *Transporte* erweitern.

Ein Transport ist ein Objekt, das die eigentliche Übertragung von Ajax-Daten behandelt. Neue Transporte werden als Factory definiert, die ein Objekt mittels der `.send()`- und `.abort()`-Methoden zurückgeben. Die `.send()`-Methode ist zuständig für das Senden der Anfrage, für den Umgang mit der Antwort und das Zurücksenden der Daten mit Callback-Funktion. Die Methode `.abort()` dient dem sofortigen Abbruch der Anfrage.

Ein eigener Transport kann beispielweise `<img>`-Elemente einsetzen, um externe Daten zu beschaffen. Dadurch kann das Laden von Bildern wie andere

Ajax-Anfragen behandelt werden und der Code ist konsistenter. Das hierzu erforderliche JavaScript ist ein wenig komplizierter, weshalb wir uns das Endprodukt ansehen und dann die Komponenten besprechen:

```
$.ajaxTransport('img', function(settings) {
    var $img, img, prop;
    return {
        send: function(headers, complete) {
            function callback(success) {
                if (success) {
                    complete(200, 'OK', {img: img});
                } else {
                    $img.remove();
                    complete(404, 'Not Found');
                }
            }
            $img = $('<img>', {
                src: settings.url
            });
            img = $img[0];
            prop = typeof img.naturalWidth === 'undefined' ?
                'width' : 'naturalWidth';
            if (img.complete) {
                callback( !!img[prop] );
            } else {
                $img.bind('load error', function(event) {
                    callback(event.type === 'load');
                });
            }
        },
        abort: function() {
            if ($img) {
                $img.remove();
            }
        }
    };
});
```

**Listing 13-17**

Wenn wir den Transport definieren, übergeben wir erst einen Datentypnamen an `$.ajaxTransport()`. So teilen wir jQuery mit, wann unser Transport statt des integrierten Mechanismus eingesetzt werden soll. Dann verwenden wir eine Funktion, die das neue Transport-Objekt zurückgibt, das die entsprechenden `.send()`- und `.abort()`-Methoden enthält.

Für unseren `img`-Transport müssen die `.send()`-Methoden ein neues `<img>`-Element erzeugen, dem wir ein `src`-Attribut zuweisen. Der Wert dieses Attributs kommt aus `settings.url`, das jQuery vom `$.ajax()`-Aufruf weitergibt. Der Brow-

ser reagiert auf die Erstellung des <img>-Elements, indem er die referenzierte Bilddatei lädt, sodass wir nur feststellen müssen, wann der Vorgang abgeschlossen ist und wann wir das Callback auslösen müssen.

Der Abschluss des Ladevorgangs ist nicht ganz einfach, wenn wir eine Vielzahl von Browzern und Versionen unterstützen wollen. In manchen Browzern können wir einfach Ereignishandler für `load` und `error` anfügen. In anderen Browzern werden `load` und `error` nicht wie erwartet ausgelöst, wenn das Bild im Cache ist. Der Code in Listing 13–17 behandelt dieses unübliche Verhalten, indem er die Werte der Eigenschaften `.complete`, `.width` und `.naturalWidth` für jeden Browser entsprechend überprüft. Haben wir festgestellt, dass ein Ladevorgang abgeschlossen ist (erfolgreich oder mit Fehler), rufen wir die `callback()`-Funktion auf, die ihrerseits die `complete()`-Funktion aufruft, die an `.send()` übergeben wurde. So kann `$.ajax()` auf den Ladevorgang reagieren.

Der Umgang mit abgebrochenen Ladevorgängen ist wesentlich einfacher. Unsere `.abort()`-Methode muss nur nach `.send()` aufräumen, indem sie das <img>-Element entfernt, sollte es erstellt worden sein.

Jetzt benötigen wir den `$.ajax()`-Aufruf, der unseren neuen Transport verwendet:

```
$(document).ready(function() {
    $.ajax({
        url: 'missing.jpg',
        dataType: 'img'
    }).done(function(img) {
        $('<div></div>', {
            id: 'picture',
            html: img
        }).appendTo('body');
    }).fail(function(xhr, textStatus, msg ) {
        $('<div></div>', {
            id: 'picture',
            html: textStatus + ': ' + msg
        }).appendTo('body');
    });
});
```

**Listing 13–18**

Um einen bestimmten Transport zu verwenden, benötigt `$.ajax()` einen passenden `dataType`-Wert. Die Handler für Erfolg oder Fehler müssen berücksichtigen, welche Art Daten ihnen übergeben worden sind. Unser `img`-Transport gibt ein <img>-DOM-Element zurück, wenn der Vorgang erfolgreich ist, sodass unser `.done()`-Handler das Element als HTML-Inhalt eines neu erstellten <div>-Elements einsetzt, das ins Dokument eingefügt wird.

In unserem Beispiel ist die angegebene Bilddatei (`missing.jpg`) nicht vorhanden. Diesen Fall fangen wir mit einem passenden `.fail()`-Handler ab, der im <div>-Element an der Stelle des Bildes eine Fehlermeldung einfügt:



Wir können diesen Fehler beheben, indem wir ein vorhandenes Bild referenzieren:

```
$(document).ready(function() {
    $.ajax({
        url: 'sunset.jpg',
        dataType: 'img'
    }).done(function(img) {
        $('

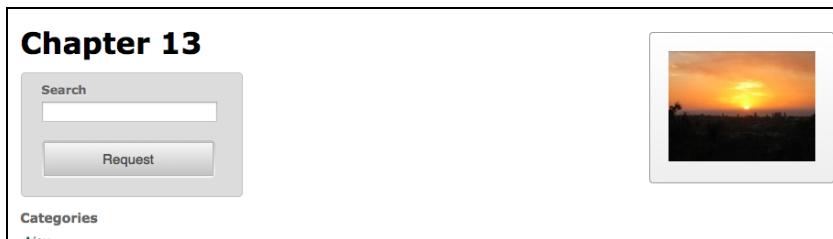
</div>', {
            id: 'picture',
            html: img
        }).appendTo('body');
    }).fail(function(xhr, textStatus, msg ) {
        $('

</div>', {
            id: 'picture',
            html: textStatus + ': ' + msg
        }).appendTo('body');
    });
});


```

***Listing 13-19***

Jetzt ist unser Transport in der Lage, das Bild erfolgreich zu laden, und wir sehen das Ergebnis auf der Seite:



Einen neuen Transport zu erstellen ist eigentlich nicht notwendig, doch selbst in diesem Fall können jQuerys Ajax-Funktionen an unsere Bedürfnisse angepasst werden.

## 13.6 Zusammenfassung

In diesem letzten Kapitel haben wir das jQuery-Ajax-Framework genau unter die Lupe genommen. Wir sind jetzt in der Lage, auf einer einzelnen Seite ein durchgängiges Benutzererlebnis zu bieten, indem wir externe Ressourcen bei Bedarf laden, und dabei auf Fehler reagieren zu können sowie Caching und Drosselung zu unterstützen. Wir haben gelernt, wie die internen Operationen des Ajax-Frameworks, wie Promises, Transporte, Prefilter und Konverter, funktionieren und wie wir diese Mechanismen für unsere Skripte erweitern können.

### 13.6.1 Literatur

Eine vollständige Liste der verfügbaren Ajax-Methoden finden Sie in Anhang C dieses Buches, im *jQuery Reference Guide* und in der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

## 13.7 Übungsaufgaben

Um die folgenden Übungsaufgaben durchführen zu können, benötigen Sie die Datei `index.html` für dieses Kapitel sowie den fertigen JavaScript-Code aus `complete.js`. Diese Dateien können Sie von der Website von dpunkt unter [www.dpunkt.de/jquery](http://www.dpunkt.de/jquery) herunterladen.

»Schwierige« Aufgaben erfordern unter Umständen die Nutzung der offiziellen jQuery-Dokumentation unter <http://api.jquery.com/>.

1. Ändern Sie die Funktion `buildItem()` so, dass sie die lange Beschreibung jeder angezeigten jQuery-Methode verwendet.
2. *Schwierig:* Bauen Sie in die Seite ein Formular ein, das auf eine öffentliche Flickr-Fotosuche zeigt (<http://www.flickr.com/search/>), und stellen Sie sicher, dass es `<input name="q">` und eine Senden-Schaltfläche besitzt. Verwenden Sie die automatische Ergänzung, um die Fotos vom JSONP-Service unter [http://api.flickr.com/services/feeds/photos\\_public.gne](http://api.flickr.com/services/feeds/photos_public.gne) abzurufen, und fügen Sie sie im Inhaltsbereich der Seite ein. Wenn Sie `data` an diesen Service senden, verwenden Sie `tags` statt `q` und setzen Sie `format` auf `json`. Beachten Sie, dass der Service ein JSONP-Callback mit dem Namen `jsoncallback` erwartet.
3. *Schwierig:* Bauen Sie einen Fehlerhandler in die Flickr-Anfrage ein für den Fall, dass es zu einem `parsererror` kommt. Testen Sie ihn, indem Sie statt des Callback-Namens `jsoncallback` wieder `callback` verwenden.

# A JavaScript-Closures

Im gesamten Buch sind uns viele jQuery-Methoden begegnet, die Funktionen als Parameter verwenden können. Unsere Beispiele haben daher oft Funktionen erzeugt, aufgerufen oder übergeben. Obwohl hierfür eigentlich schon ein grobes Verständnis der internen JavaScript-Mechanismen ausreicht, sind die Nebenwirkungen dieser Aktionen manchmal sehr eigenartig, wenn wir die Funktionen der Sprache nicht besser kennen. In diesem Anhang zeigen wir eines der funktionsbasierten Konstrukte namens *Closure*, das eher etwas für Eingeweihte (trotzdem aber geläufig) ist.

Wir werden viele kleine Codebeispiele behandeln, in denen wir eine Reihe von Meldungen ausgeben. Statt einen browserspezifischen Logging-Mechanismus zu verwenden (wie `console.log()`) oder eine Reihe von `alert()`-Dialogen, nutzen wir wie folgt eine kleine Plug-in-Methode:

```
$ .print = function(message) {  
    $(document).ready(function() {  
        $('<div class="result"><div>').  
            .text(String(message))  
            .appendTo('#results');  
    });  
};
```

Mit dieser Methode rufen wir `$.print('hello')` auf, um die Meldung innerhalb von `<div id="results">` einzufügen.

## A.1 Innere Funktionen

Erfreulicherweise gehört JavaScript zu den Programmiersprachen, die die Deklaration *innerer Funktionen* unterstützen. Viele traditionelle Programmiersprachen, wie C, sammeln Funktionen in einer höheren Ebene. Sprachen mit inneren Funktionen ermöglichen uns deren Deklaration dort, wo sie tatsächlich benötigt werden, was unseren Namensraum sauber hält.

Eine innere Funktion ist einfach eine Funktion, die innerhalb einer anderen definiert wird. Zum Beispiel:

```
function outerFn() {
    function innerFn() {
    }
}
```

Hierbei ist `innerFn()` die innere Funktion, die in `outerFn()` enthalten ist. Das bedeutet, dass ein Aufruf von `innerFn()` innerhalb von `outerFn()` zulässig ist, aber nicht von außerhalb. Der folgende Code erzeugt einen JavaScript-Fehler:

```
function outerFn() {
    $.print('Outer function');
    function innerFn() {
        $.print('Inner Function');
    }
}
$.print('innerFn():');
innerFn();
```

Wir können den Code dennoch erfolgreich ausführen, wenn wir die `innerFn()` innerhalb von `outerFn()` ausführen, wie folgt:

```
function outerFn() {
    $.print('Outer function');
    function innerFn() {
        $.print('Inner function');
    }
    innerFn();
}
$.print('outerFn():');
outerFn();
```

Dadurch entsteht folgende Ausgabe:

```
outerFn():
Outer function
Inner function
```

Dieses Verfahren ist besonders bei kleinen Funktionen mit einer einzigen Verwendung sinnvoll. Rekursive Algorithmen, die einen nicht rekursiven API-Wrapper haben, werden häufig am einfachsten mit einer inneren Hilfsfunktion aufgebaut.

### A.1.1 Gesprengte Ketten

Wenn Funktionsreferenzen ins Spiel kommen, wird die Sache komplizierter. Einige Sprachen, wie Pascal, ermöglichen die Verwendung innerer Funktionen nur, um Code zu verbergen. Solche Funktionen sind für immer in ihren Elternfunktionen verborgen. JavaScript ermöglicht uns dagegen, diese Funktionen zu übergeben wie jede andere Art von Daten. So können innere Funktionen ihrer Elternfunktion entkommen.

Der Fluchtweg kann verschiedene Richtungen nehmen. Stellen Sie sich vor, die Funktion wird einer *globalen Variablen* zugewiesen:

```
var globalVar;  
  
function outerFn() {  
    $.print('Outer function');  
    function innerFn() {  
        $.print('Inner function');  
    }  
    globalVar = innerFn;  
}  
$.print('outerFn():');  
outerFn();  
$.print('globalVar():');  
globalVar();
```

Der Aufruf von `outerFn()` nach der Funktionsdefinition modifiziert die globale Variable `globalVar`. Sie dient jetzt als Referenz zu `innerFn()`. Das bedeutet, dass ein späterer Aufruf von `globalVar` genauso funktioniert wie ein Aufruf von `innerFn()` und die Printanweisungen erreichbar sind:

```
outerFn():  
Outer function  
globalVar():  
Inner function
```

Beachten Sie, dass der Aufruf von `innerFn()` außerhalb von `outerFn()` immer noch zu einem Fehler führt! Obwohl die Funktion jedoch über die Referenz in der globalen Variablen entkommen konnte, ist der Funktionsname immer noch innerhalb des Gültigkeitsbereichs von `outerFn()` gefangen.

Eine Funktionsreferenz kann ihren Weg aus der Elternfunktion heraus auch mittels eines *Rückgabewerts* finden:

```
function outerFn() {  
    $.print('Outer function');  
    function innerFn() {  
        $.print('Inner function');  
    }  
    return innerFn;  
}  
$.print('var fnRef = outerFn():');  
var fnRef = outerFn();  
$.print('fnRef():');  
fnRef();
```

Hier gibt es keine globale Variable, die in `outerFn()` modifiziert wird. Stattdessen übergibt `outerFn()` eine Referenz an `innerFn()`. Der Aufruf von `outerFn()` resultiert in dieser Referenz, die gespeichert und selbst wieder aufgerufen wird, was zur Ausgabe folgender Meldung führt:

```
var fnRef = outerFn():
Outer function
fnRef():
Inner function
```

Die Tatsache, dass innere Funktionen durch eine Referenz aufgerufen werden können, auch wenn die Funktion ihren Gültigkeitsbereich verlassen hat, bedeutet, dass JavaScript referenzierte Funktionen so lange verfügbar halten muss, wie sie aufgerufen werden können. Jede Variable, die auf eine Funktion zeigt, wird von der JavaScript *Runtime* beobachtet, und wenn die letzte verschwunden ist, gibt der JavaScript *Garbage Collector* dieses Speicherhäppchen frei.

### A.1.2 Gültigkeitsbereiche von Variablen

Innere Funktionen können natürlich auch ihre eigenen Variablen haben, die auf den Gültigkeitsbereich der Funktion eingeschränkt sind:

```
function outerFn() {
    function innerFn() {
        var innerVar = 0;
        innerVar++;
        $.print('innerVar = ' + innerVar);
    }
    return innerFn;
}
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

Jedes Mal, wenn diese innere Funktion aufgerufen wird, per Referenz oder anderswie, wird eine neue Variable namens `innerVar` erstellt, hochgezählt und wie folgt angezeigt:

```
innerVar = 1
innerVar = 1
innerVar = 1
innerVar = 1
```

Innere Funktionen können globale Variablen genauso wie jede andere Funktion referenzieren:

```
var globalVar = 0;
function outerFn() {
    function innerFn() {
        globalVar++;
        $.print('globalVar = ' + globalVar);
```

```
        }
        return innerFn;
    }
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

Jetzt zählt unsere Funktion die Variable bei jedem Aufruf ordnungsgemäß hoch:

```
globalVar = 1
globalVar = 2
globalVar = 3
globalVar = 4
```

Was passiert aber, wenn die Variable lokal zur Elternfunktion gehört? Da die innere Funktion den Gültigkeitsbereich der Elternfunktion erbt, kann diese Variable ebenfalls referenziert werden:

```
function outerFn() {
    var outerVar = 0;
    function innerFn() {
        outerVar++;
        $.print('outerVar = ' + outerVar);
    }
    return innerFn;
}
var fnRef = outerFn();
fnRef();
fnRef();
var fnRef2 = outerFn();
fnRef2();
fnRef2();
```

Jetzt wird das Verhalten unserer Funktion interessanter:

```
outerVar = 1
outerVar = 2
outerVar = 1
outerVar = 2
```

Dieses Mal erhalten wir eine Mischung aus zwei früheren Effekten. Die Aufrufe von `innerFn()` durch jede Referenzierung zählen `outerVar` unabhängig hoch. Der zweite Aufruf von `outerFn()` setzt den Wert von `outerVar` nicht zurück, sondern erzeugt eine neue Instanz von `outerVar`, die auf den Gültigkeitsbereich des zweiten Funktionsaufrufs beschränkt ist. Das Ergebnis ist, dass nach den vorherigen Aufrufen ein weiterer Aufruf von `fnRef()` den Wert 3 ergibt und der folgende Aufruf von `fnRef2()` ebenfalls 3. Die beiden Zähler verhalten sich unabhängig voneinander.

Wenn eine Referenz auf eine innere Funktion ihren Weg aus dem Gültigkeitsbereich der Funktion, in der sie definiert wurde, herausfindet, entsteht eine *Closure* dieser Funktion. Wir nennen Variablen, die weder Parameter darstellen noch lokal in der inneren Funktion sind, *freie Variablen*, und die Umgebung der äußeren Funktion umschließt sie. Die Tatsache, dass die Funktion eine lokale Variable in der äußeren Funktion referenziert, ermöglicht ihre Ausführung. Der Speicher wird nicht freigegeben, wenn die Funktion abgeschlossen ist, da er von der Closure noch benötigt wird.

## A.2 Interaktion zwischen Closures

Wenn mehr als eine innere Funktion vorhanden ist, können Closures schwer vorherzusagende Auswirkungen haben. Nehmen wir an, wir koppeln unsere Inkrementfunktion mit einer weiteren, die jeweils um zwei hochzählt:

```
function outerFn() {
    var outerVar = 0;
    function innerFn1() {
        outerVar++;
        $.print('(1) outerVar = ' + outerVar);
    }
    function innerFn2() {
        outerVar += 2;
        $.print('(2) outerVar = ' + outerVar);
    }
    return {'fn1': innerFn1, 'fn2': innerFn2};
}
var fnRef = outerFn();
fnRef.fn1();
fnRef.fn2();
fnRef.fn1();
var fnRef2 = outerFn();
fnRef2.fn1();
fnRef2.fn2();
fnRef2.fn1();
```

Wir übergeben die Referenzen auf beide Funktionen mit einer *Map* (was eine andere Vorgehensweise zeigt, wie eine innere Funktion ihrem Elternelement entkommen kann). Beide Funktionen werden über die Referenz aufgerufen:

```
(1) outerVar = 1
(2) outerVar = 3
(1) outerVar = 4
(1) outerVar = 1
(2) outerVar = 3
(1) outerVar = 4
```

Die beiden inneren Funktionen beziehen sich auf dieselbe lokale Variable, sodass sie sich in der gleichen Closure befinden. Wenn `innerFn1()` `outerVar` um 1 erhöht, wird dadurch der neue Startwert für `outerVar` gesetzt, wenn `innerFn2()` aufgerufen wird, und umgekehrt. Wieder einmal sehen wir jedoch, dass alle folgenden Aufrufe von `outerFn()` neue Instanzen von Closures mit eigenen, passenden Umgebungen erzeugen. Wenn Sie sich mit objektorientierter Programmierung auskennen, sehen Sie, dass wir faktisch ein neues Objekt erstellt haben, dessen freie Variablen als *Instanzvariablen* und dessen Closures als *Instanzmethoden* fungieren. Die Variablen sind *privat*, da sie nicht direkt von außerhalb der Closure referenziert werden können, was objektorientierten privaten Daten gleichkommt.

## A.3 Closures in jQuery

Die Methoden, die wir in der jQuery-Bibliothek kennengelernt haben, verwenden oft zumindest eine Funktion als Parameter. Aus Bequemlichkeit haben wir häufig *anonyme Funktionen* verwendet, sodass wir das Verhalten der Funktion dort definieren können, wo es benötigt wird. Das bedeutet, dass sich die Funktionen selten im oberen Namensraum befinden, sondern üblicherweise innere Funktionen sind, die Closures einfach erzeugen können.

### A.3.1 Argumente für `$(document).ready()`

Beinahe aller Code, den wir in jQuery schreiben, landet in einer Funktion, die als Argument an `$(document).ready()` übergeben wird. Dadurch stellen wir sicher, dass das DOM geladen ist, bevor der Code ausgeführt wird, was eine Voraussetzung für interessanten jQuery-Code darstellt. Wenn eine Funktion erstellt und an `.ready()` übergeben wird, wird eine Referenz auf diese Funktion als Teil des globalen jQuery-Objekts gespeichert. Diese Referenz wird dann zu einem späteren Zeitpunkt aufgerufen, wenn das DOM bereit ist.

Wir platzieren das `$(document).ready()`-Konstrukt normalerweise oben in der Codestruktur, sodass diese Funktion kein Teil einer Closure ist. Da unser Code normalerweise jedoch innerhalb einer Funktion steht, ist der Rest eine innere Funktion:

```
$(document).ready(function() {
    var readyVar = 0;
    function innerFn() {
        readyVar++;
        $.print('readyVar = ' + readyVar);
    }
    innerFn();
    innerFn();
});
```

Das sieht wie viele unserer früheren Beispiele aus, nur dass in diesem Fall die äußere Funktion der Callback an `$(document).ready()` ist. Da `innerFn()` darin definiert ist und die Variable `readyVar` referenziert, die im Gültigkeitsbereich der Callback-Funktion liegt, erzeugt `innerFn()` und seine Umgebung eine Closure. Wir erkennen das am Wert von `readyVar`, der zwischen zwei Funktionsaufrufen bestehen bleibt:

```
readyVar = 1  
readyVar = 2
```

Die Tatsache, dass der meiste jQuery-Code innerhalb einer Funktion steht, verhindert Kollisionen im Namensraum. Diese Eigenschaft ermöglicht es z.B. `jQuery.noConflict()`, das Kürzel `$` für andere Bibliotheken freizugeben und dabei immer noch lokale Kürzel innerhalb von `$(document).ready()` zu definieren.

### A.3.2 Ereignishandler

Das Konstrukt `$(document).ready()` verpackt normalerweise den Rest unseres Codes einschließlich der Zuweisung von *Ereignishandlern*. Da Handler Funktionen sind, werden sie zu inneren Funktionen. Als innere Funktionen werden sie gespeichert und später aufgerufen und können Closures erzeugen. Ein einfacher `click`-Handler zeigt dies:

```
$(document).ready(function() {  
    var counter = 0;  
    $('#button-1').click(function() {  
        counter++;  
        $.print('counter = ' + counter);  
        return false;  
    });  
});
```

Da die Variable `counter` innerhalb des `.ready()`-Handlers deklariert ist, steht sie nur dem jQuery-Code innerhalb des Blocks zur Verfügung und nicht dem außerhalb. Sie kann aber vom Code des `click`-Handlers referenziert werden, der den Wert der Variablen hochzählt und anzeigt. Immer, wenn die gleiche Instanz von `counter` durch Klick auf einen Link referenziert wird, entsteht eine Closure. Das bedeutet, dass die Meldung einen sich ständig vergrößernden Wert anzeigt, und nicht jedes Mal 1:

```
counter = 1  
counter = 2  
counter = 3
```

Ereignishandler können ihre Closure-Umgebung gemeinsam nutzen, wie andere Funktionen auch:

```
$(document).ready(function() {
    var counter = 0;
    $('#button-1').click(function() {
        counter++;
        $.print('counter = ' + counter);
        return false;
    });
    $('#button-2').click(function() {
        counter--;
        $.print('counter = ' + counter);
        return false;
    });
});
```

Da beide Funktionen dieselbe Zählvariable nutzen, verändern die Inkrement- und Dekrementoperationen der beiden Links denselben Wert und nicht zwei unabhängige:

```
counter = 1
counter = 2
counter = 1
counter = 0
```

### A.3.3 Handler in Schleifen binden

Schleifenkonstruktionen können aufgrund der Funktionsweise von Closures zu interessanten Herausforderungen werden. Stellen Sie sich vor, wir erstellen in einer Schleife Elemente und binden Verhaltensweisen an diese Elemente, die vom Schleifenindex abhängen:

```
$(document).ready(function() {
    for (var i = 0; i < 5; i++) {
        $('<div>Print ' + i + '</div>')
            .click(function() {
                $.print(i);
            }).insertBefore('#results');
    }
});
```

Die Variable *i* wird nacheinander auf die Werte 0 bis 4 gesetzt und es wird jedes Mal ein neues *<div>*-Element erzeugt. Die Elemente besitzen ein eindeutiges Textlabel, wie wir es auch erwarten würden:

```
Print 0
Print 1
Print 2
Print 3
Print 4
```

Ein Klick auf eines dieser Elemente ergibt jedoch die Ausgabe der Zahl 5 und nicht der Zahl des dazugehörigen Labels, wie wir erwarten würden. Jede Referenz des click-Handlers auf i ist die gleiche. Auch wenn der Wert von i zu dem Zeitpunkt des Bindens anders ist, ist die Variable dieselbe und daher wird der Endwert von i (5) verwendet, wenn ein Klick stattfindet.

Wir können dieses Problem auf verschiedene Weise umgehen. Zuerst könnten wir die for-Schleife durch die jQuery-Funktion \$.each() wie folgt ersetzen:

```
$(document).ready(function() {
    $.each([0, 1, 2, 3, 4], function(index, value) {
        $('<div>Print ' + value + '</div>')
            .click(function() {
                $.print(value);
            }).insertBefore('#results');
    });
});
```

Funktionsparameter werden wie Variablen innerhalb von Funktionen definiert: Der Variablenwert ist faktisch bei jedem Durchlauf der Schleife ein anderer. Dies führt dazu, dass jeder click-Handler auf eine andere Variable zeigt und Klicks auf die Elemente Zahlen ausgeben, die dem Label wie geplant entsprechen.

Wir können dieselben Eigenschaften der Funktionsparameter jedoch auch nutzen, um das Problem ohne den Aufruf von \$.each() zu lösen. In der Schleife können wir eine neue Funktion definieren und ausführen, die für die Separation der i-Werte in verschiedene Variablen sorgt:

```
$(document).ready(function() {
    for (var i = 0; i < 5; i++) {
        (function(value) {
            $('<div>Print ' + value + '</div>')
                .click(function() {
                    $.print(value);
                }).insertBefore('#results');
        })(i);
    }
});
```

Wir haben dieses Konstrukt namens *Immediately Invoked Function Expression* (IIFE) bereits als Mittel zur Umbenennung des \$-Alias für das jQuery-Objekt nach Aufruf von \$.nonConflict() kennengelernt. Hier verwenden wir es, um i als Parameter namens value an den click-Handler zu übergeben.

Schließlich verwenden wir eine Funktion des jQuery-Ereignissystems, um das Problem auf andere Weise zu lösen. Die Methode .bind() akzeptiert einen Objekt-parameter, der mit dem Ereignishandler als event.data übergeben wird:

```
$(document).ready(function() {
    for (var i = 0; i < 5; i++) {
        $('<div>Print ' + i + '</div>')
            .bind('click', {value: i}, function(event) {
                $.print(event.data.value);
            }).insertBefore('#results');
    }
});
```

In diesem Fall wird `i` als Datum an die `.bind()`-Methode übergeben und kann innerhalb des Handlers über eine Prüfung von `event.data.value` abgerufen werden. Da `event` auch hier wieder ein Funktionsparameter ist, bildet es bei jedem Aufruf des Handlers eine eindeutige Einheit (unique entity) und keinen separaten Wert, der von allen genutzt wird.

#### A.3.4 Benannte und anonymous Funktionen

Diese Beispiele haben anonymous Funktionen verwendet, wie wir das bislang im jQuery-Code immer getan haben. In der Konstruktion der Closures macht das keinen Unterschied, sie können aus benannten oder anonymen Funktionen gebildet werden. Zum Beispiel können wir eine anonymous Funktion schreiben, die den Index eines Elements innerhalb eines jQuery-Objekts zurückgibt:

```
$(document).ready(function() {
    $('input').each(function(index) {
        $(this).click(function() {
            $.print('index = ' + index);
            return false;
        });
    });
});
```

Da die innerste Funktion innerhalb des `.each()`-Callbacks definiert ist, erzeugt dieser Code so viele Funktionen wie Schaltflächen vorhanden sind. Jede der Funktionen ist als `click`-Handler mit einer der Schaltflächen verbunden. In ihrer Closing-Umgebung verwenden die Funktionen `index`, da dies der Parameter für den `.each()`-Callback ist. Das Verhalten ist genauso, als wenn der `click`-Handler als benannte Funktion geschrieben wäre:

```
$(document).ready(function() {
    $('input').each(function(index) {
        function clickHandler() {
            $.print('index = ' + index);
            return false;
        }

        $(this).click(clickHandler);
    });
});
```

Die Version mit der anonymen Funktion ist einfach nur ein bisschen kürzer. Die Position der benannten Funktion ist aber noch immer von Bedeutung:

```
$(document).ready(function() {
    function clickHandler() {
        $.print('index = ' + index);
        return false;
    }

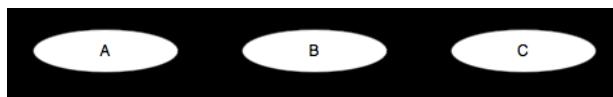
    $('input').each(function(index) {
        $(this).click(clickHandler);
    });
});
```

Diese Version löst einen JavaScript-Fehler aus, wenn eine Schaltfläche angeklickt wird, da sich `index` nicht in der Closure von `clickHandler()` befindet. Es bleibt eine freie Variable und ist in diesem Kontext nicht definiert.

## A.4 Gefahren durch Speicherlecks

JavaScript verwaltet seinen Speicher mit einem Verfahren, das als Garbage Collection bekannt ist. Das steht im Gegensatz zu einfachen Programmiersprachen wie C, die von Programmierern erwarten, Speicherblöcke explizit zu reservieren und wieder freizugeben, wenn sie nicht mehr benötigt werden. Andere Sprachen, wie Objective-C, unterstützen den Programmierer bei der Implementierung eines Referenzzählersystems, das es dem Benutzer ermöglicht, festzuhalten, wie viele Teile des Programms einen bestimmten Speicherbereich verwenden, sodass er freigegeben werden kann, wenn er nicht mehr benötigt wird. JavaScript ist eine weiterentwickelte Programmiersprache und erledigt all diese Aufgaben normalerweise im Hintergrund.

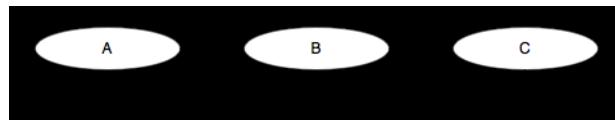
Immer wenn ein neues speicherresidentes Element wie ein Objekt oder eine Funktion im JavaScript-Code auftaucht, wird ein Stückchen Speicher dafür reserviert. Während das Objekt Funktionen übergeben und Variablen zugewiesen wird, zeigen immer mehr Codeabschnitte auf dieses Objekt. JavaScript verwaltet diese *Zeiger*, und wenn der letzte davon verschwunden ist, wird der vom Objekt genutzte Speicher freigegeben. Stellen Sie sich eine Kette von Zeigern wie im folgenden Diagramm vor:



Hier hat Objekt A eine Eigenschaft, die auf B zeigt, und B hat eine Eigenschaft, die auf C zeigt. Auch wenn Objekt A hier nur eine Variable im aktuellen Gültigkeitsbereich ist, müssen alle drei Objekte im Speicher bleiben, da es die Zeiger auf

sie gibt. Wenn A jedoch aus seinem Gültigkeitsbereich herauskommt (z.B. am Ende der Funktion, in der A deklariert wurde), kann es vom Garbage Collector freigegeben werden. Jetzt zeigt nichts mehr auf B, sodass auch B freigegeben werden kann, und schließlich auch C.

Mit komplexeren Referenzen ist der Umgang nicht mehr so leicht:



Jetzt haben wir Objekt C eine Eigenschaft hinzugefügt, die sich auf B bezieht. Wenn A freigegeben wird, hat B in diesem Fall von C noch immer einen Zeiger auf sich gerichtet. Diese Verweisschleife muss durch JavaScript gesondert behandelt werden, das erkennen muss, wie die gesamte Schleife isoliert von den Variablen im Gültigkeitsbereich zu sehen ist.

#### A.4.1 Unerwünschte Verweisschleifen

Closures können zu unerwünschten Verweisschleifen führen. Da Funktionen Objekte darstellen, die im Speicher gehalten werden müssen, werden alle Variablen, die sich in deren Closing-Umgebung befinden, auch im Speicher behalten:

```
function outerFn() {
    var outerVar = {};
    function innerFn() {
        $.print(outerVar);
    }
    outerVar.fn = innerFn;
    return innerFn;
};
```

Hier wird ein Objekt namens `outerVar` erzeugt und aus der inneren Funktion `innerFn()` referenziert. Dann wird eine Eigenschaft von `outerVar` erstellt, die auf `innerFn()` zeigt, und `innerFn()` zurückgegeben. Dies erzeugt eine Closure für `innerFn()`, die sich auf `outerVar` bezieht, die wiederum `innerFn()` referenziert.

Die Schleife kann sogar noch tückischer werden:

```
function outerFn() {
    var outerVar = {};
    function innerFn() {
        $.print('hello');
    }
    outerVar.fn = innerFn;
    return innerFn;
};
```

Hier haben wir `innerFn()` verändert, sodass sie `outerVar` nicht mehr referenziert. Dadurch wird die Schleife aber nicht durchbrochen! Auch wenn `outerVar` niemals von `innerFn()` referenziert wird, befindet sie sich trotzdem in der Closing-Umgebung von `innerFn()`. Alle Variablen im Gültigkeitsbereich von `outerFn()` werden durch die Closure *implizit* durch `innerFn()` referenziert. Daher führen Closures leicht zu unbeabsichtigten Schleifen dieser Art.

#### A.4.2 Internet Explorer und sein Speicherleck-Problem

Normalerweise stellt das kein Problem dar, da JavaScript diese Schleifen erkennt und sie aufräumt, wenn sie zurückbleiben. Einige Versionen des Internet Explorers (IE) jedoch haben Schwierigkeiten damit, eine bestimmte Klasse von Verweisschleifen zu behandeln. Wenn eine Schleife sowohl DOM-Elemente als auch reguläre JavaScript-Objekte enthält, kann IE keine davon freigeben, da sie durch unterschiedliche Speichermanager behandelt werden. Diese Schleifen werden erst freigegeben, wenn der Browser geschlossen wird, was über seine Laufzeit zu einem großen Speicherverbrauch führen kann. Eine solche Schleife entsteht leicht durch einen einfachen Ereignishandler wie diesen:

```
$(document).ready(function() {
    var button = document.getElementById('button-1');
    button.onclick = function() {
        $.print('hello');
        return false;
    };
});
```

Wenn der `click`-Handler zugewiesen wird, entsteht eine Closure, in der `button` enthalten ist. `button` enthält jetzt eine Referenz in die Closure, nämlich die Eigenschaft `onclick`. Die sich dadurch ergebende Schleife kann vom Internet Explorer nicht freigegeben werden, selbst wenn wir die Seite verlassen.

Um den Speicher wieder freizugeben, müssten wie die Schleife durchbrechen, z.B. indem wir die `onclick`-Eigenschaft loswerden, bevor das Fenster geschlossen wird (wobei wir aufpassen müssen, dass wir keine neue Schleife zwischen `window` und seinem `onunload`-Handler erzeugen). Eine Alternative besteht darin, den Code ohne Closure umzuschreiben:

```
function hello() {
    $.print('hello');
    return false;
}
$(document).ready(function() {
    var button = document.getElementById('button-1');
    button.onclick = hello;
});
```

Da die `hello()`-Funktion nicht mehr über `button` geschlossen wird, zeigt die Referenz nur in eine Richtung (von `button` zu `hello`), es gibt keine Schleife und damit auch kein Speicherleck.

### Die gute Nachricht

Jetzt schreiben wir denselben Code, aber mit normalen jQuery-Konstrukten:

```
$(document).ready(function() {
    var $button = $('#button-1');
    $button.click(function() {
        $.print('hello');
        return false;
    });
});
```

Auch wenn noch immer eine Closure entsteht, und damit eine Schleife wie vorher, bekommen wir mit diesem Code kein IE-Speicherleck. Glücklicherweise kennt jQuery diese potenziellen Speicherlecks und gibt alle zugewiesenen Ereignishandler manuell wieder frei. Solange wir vertrauensvoll die jQuery-Methoden zum Binden von Ereignissen für unsere Handler einsetzen, müssen wir uns keine Sorgen um Speicherlecks dieses speziellen Typs machen.

Damit sind wir aber nicht komplett aus dem Schneider. Wir müssen vorsichtig sein, wenn wir mit DOM-Elementen andere Aufgaben durchführen. JavaScript-Objekte an DOM-Elemente anzufügen kann trotzdem im Internet Explorer zu Speicherlecks führen. jQuery hilft uns jedoch dabei, dass diese Situation weniger häufig eintritt.

Als Folge bietet uns jQuery ein weiteres Werkzeug, diese Lecks zu vermeiden. In Kapitel 12, »*DOM-Manipulation für Fortgeschrittene*«, haben wir gesehen, dass uns die `.data()`-Methode ermöglicht, Informationen an DOM-Elemente anzufügen, wie wir es auch mit *expando*-Eigenschaften machen würden. Da diese Daten nicht als expando gespeichert werden (jQuery verwendet eine interne Map und legt die Daten anhand von IDs ab), ergibt sich eine solche Referenzschleife nicht und wir umgehen das Speicherleck-Problem. Immer wenn uns *expando* als bequeme Möglichkeit erscheint, Daten zu speichern, sollten wir `.data()` als sicherere Alternative einsetzen.

## A.5 Zusammenfassung

Closures sind eine leistungsfähige Eigenschaft von JavaScript. Sie sind häufig sinnvoll, wenn Variablen vor anderem Code versteckt werden sollen, sodass wir nicht über anderswo verwendete Variablennamen stolpern. Da jQuery oftmals Funktionen als Methodenargumente einsetzt, können sie häufiger unbeabsichtigt entstehen. Unser Verständnis von Closures ermöglicht es uns, effizienteren und kürzeren Code zu schreiben, und mit ein wenig Vorsicht und der Verwendung der in jQuery integrierten Sicherheitsmechanismen können wir Speicherlecks und ihre Folgen vermeiden.

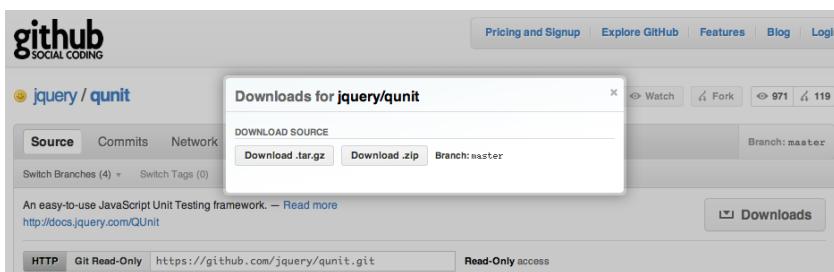
## B JavaScript mit QUnit testen

In diesem Buch haben wir eine Menge JavaScript-Code geschrieben und gesehen, wie uns jQuery dabei hilft, den Code relativ bequem zu erstellen. Immer wenn wir eine Funktion eingebaut haben, mussten wir jedoch in einem weiteren Schritt manuell prüfen, ob auf unserer Website noch alles wie gewünscht ablief. Bei einfachen Aufgaben mag das noch angehen, bei großen und komplexen Projekten wird das manuelle Testen aber lästig. Neue Anforderungen können rückwirkend zu Fehlern führen, die Teile unsere Skripte betreffen, die vorher noch einwandfrei funktionierten. Diese Fehler sind nur allzu leicht zu übersehen, da sie nicht in den jüngsten Änderungen im Code liegen, die wir natürlich zuerst testen würden.

Wir benötigen demzufolge ein automatisiertes System, das diese Tests für uns ausführt. Das *QUnit*-Testframework ist ein solches System. Obwohl es verschiedene Testframeworks mit unterschiedlichen Vorzügen gibt, empfehlen wir QUnit für die meisten jQuery-Projekte, da es vom jQuery-Projekt entwickelt wurde und gewartet wird. jQuery nutzt QUnit sogar selbst (mit beinahe 5000 Tests).

### B.1 QUnit herunterladen

Am einfachsten finden Sie QUnit im GitHub-Repository unter <https://github.com/jquery/qunit/>. Wenn Sie nicht wissen, wie Sie Git verwenden, oder kein GitHub-Account besitzen, um QUnit herunterzuladen, klicken Sie einfach auf die Download-Schaltfläche und wählen Sie entweder die .tar.gz- oder .zip-Datei aus, um sie auf Ihrem Computer zu speichern, wie im folgenden Screenshot gezeigt:



Wenn die Datei heruntergeladen ist, entpacken (entzippen) Sie die archivierten Dateien und kopieren die Dateien `qunit.js` und `qunit.css` aus dem `qunit`-Unterordner in einen Ordner innerhalb Ihres eigenen Projekts.

## B.2 Das Dokument einrichten

Wenn die QUnit-Dateien an Ort und Stelle sind, können wir das HTML-Dokument einrichten und testen. In einem typischen Projekt würden wir die Datei `index.html` nennen und im selben Unterordner wie `qunit.js` und `qunit.css` ablegen. Für unsere Demonstration legen wir sie in einem übergeordneten Verzeichnis ab.

Der `<head>` des Dokuments enthält ein `<link>`-Tag für die CSS-Datei und `<script>`-Tags für jQuery, QUnit und das zu testende JavaScript (`B.js`) sowie die Tests selbst (`test/test.js`). Das `<body>`-Tag enthält sechs Hauptelemente, die alle mit einer ID ausgezeichnet sind, die QUnit zum Ausführen und zum Anzeigen der Testergebnisse verwendet. Eine Datei mit ähnlichen Auszeichnungen, die sich im `Test`-Ordner des heruntergeladenen Archivs befindet, kann für die ersten Schritte als Vorlage verwendet werden.

Um QUnit zu demonstrieren, verwenden wir Teile aus Kapitel 2, »Elemente auswählen«, und aus Kapitel 6, »Daten mit Ajax senden«:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>Appendix B Tests</title>
  <link rel="stylesheet" href="qunit.css" media="screen">

  <script src="jquery.js"></script>
  <script src="test/qunit.js"></script>
  <script src="B.js"></script>
  <script src="test/test.js"></script>

</head>
<body>
  <h1 id="qunit-header">Appendix B Tests</h1>
  <h2 id="qunit-banner"></h2>
  <div id="qunit-testrunner-toolbar"></div>
  <h2 id="qunit-userAgent"></h2>
  <ol id="qunit-tests"></ol>
  <div id="qunit-fixture">

    <!-- Test Markup Goes Here -->

  </div>
</body>
</html>
```

Da der zu testende Code aus Kapitel 2 auf dem DOM basiert, sollte unser Test-Markup dem auf der tatsächlichen Seite entsprechen. Wir können den HTML-Inhalt einfach aus Kapitel 2 kopieren und einfügen und ersetzen dabei den Kommentar <!--Test Markup Goes Here -->.

## B.3 Tests organisieren

QUnit bietet zwei Arten der Gruppierung, die nach ihren Funktionsaufrufen benannt sind: `module()` und `test()`. *Module* verhält sich wie eine allgemeine Test-kategorie, unter der die Tests ausgeführt werden. *Test* ist eigentlich ein Satz von Tests, der in einem Callback alle spezifischen *Unit Tests* durchführt. Wir gruppieren unsere Tests nach dem Kapiteltitel und platzieren den Code in die Datei `test/test.js`:

```
module('Selecting');

test('Child Selector', function() {
    // Hier folgt der Test
});

test('Attribute Selectors', function() {
    // Hier folgt der Test
});

module('Ajax')
```

Es ist nicht erforderlich, die Datei mit der Teststruktur einzurichten, Sie sollten die allgemeine Struktur jedoch im Kopf haben. Beachten Sie, dass unsere *Modules* und *Tests* nicht innerhalb eines `$(document).ready()`-Aufrufs enthalten sein müssen, da QUnit per Voreinstellung darauf wartet, dass das Fenster aufgebaut ist, bis es die Tests ausführt. Mit diesem einfachen Setup sehen die Testergebnisse nach dem Laden des Test-HTML aus, wie in der folgenden Seite dargestellt:

The screenshot shows a QUnit test results page. At the top, it says "Appendix B Tests" followed by "noglobals notrycatch". Below that is a "Hide passed tests" checkbox. The main area has a blue header bar with the text "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:5.0) Gecko/20100101 Firefox/5.0". Underneath, it says "Tests completed in 21 milliseconds. 0 tests of 0 passed, 0 failed." There are two numbered sections: "1. Selecting: Child Selector (0, 0, 0) Rerun" and "2. Selecting: Attribute Selectors (0, 0, 0) Rerun".

Der Modulname ist hellblau und der Testname etwas dunkler. Ein Klick darauf öffnet die Ergebnisse dieses Testlaufs, die per Voreinstellung eingeklappt sind,

wenn alle Tests erfolgreich bestanden wurden (oder wie in diesem Fall keine Tests durchgeführt wurden). Das Ajax-Modul erscheint nicht, da wir dafür noch keine Tests geschrieben haben.

## B.4 Tests hinzufügen und ausführen

Bei der testgestützten Entwicklung (Test-Driven Development, TDD) schreiben wir die Tests vor dem Code. So können wir beobachten, ob ein Test fehlschlägt, neuen Code hinzufügen und dann sehen, ob der Test bestanden wird, was bedeutet, dass unser Code wie gewünscht funktioniert.

Wir beginnen mit einem Test des Kindselektors, den wir in Kapitel 2 verwendet haben, um die Klasse `horizontal` allen `<li>`-Elementen hinzuzufügen, die Kinder von `<ul id="selected-plays">` sind:

```
test('Child Selector', function() {
  expect(1);
  var topLis = $('#selected-plays > li.horizontal');
  equal(topLis.length, 3, 'Top LIs have horizontal class');
});
```

Faktisch haben wir zwei Tests eingefügt. Wir beginnen mit den `expect()`-Test, der QUnit mitteilt, wie viele Tests wir in einem Testlauf erwarten. Da wir testen, welche Elemente auf der Seite ausgewählt werden können, verwenden wir dann den `equal()`-Test, um die Anzahl der `<li>`-Elemente der oberen Ebene mit der Zahl 3 zu vergleichen. Sind beide gleich, ist der Test bestanden:

The screenshot shows the QUnit test results for the 'Child Selector' test. The test was run on Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:5.0) Gecko/20100101 Firefox/5.0. The test completed in 22 milliseconds, with 0 tests passed and 1 failed. The failed test, '1. Top LIs have horizontal class', has a red background. It shows the expected value was 3, the result was 0, and the difference was 3. The source code for this test is located at line 102 of qunit.js. There are also links for 'Rerun' and another test, '2. Selecting: Attribute Selectors'.

Appendix B Tests	
■ noglobals ■ notrycatch	
<input type="checkbox"/> Hide passed tests	
Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:5.0) Gecko/20100101 Firefox/5.0	
Tests completed in 22 milliseconds.	
0 tests of 1 passed, 1 failed.	
1. Selecting: Child Selector (1, 0, 1) Rerun	
1. Top LIs have horizontal class	
Expected:	3
Result:	0
Diff:	3 0
Source: ()@http://book.dev/6549/B/test/qunit.js:102	
2. Selecting: Attribute Selectors (0, 0, 0) Rerun	

Natürlich schlägt dieser Test fehl, da wir den Code zum Hinzufügen der Klasse `horizontal` noch nicht geschrieben haben. Allerdings ist es einfach, diesen Code

hinzuzufügen. Wir verwenden hierzu die Hauptskriptdatei der Seite, die wir B.js genannt haben:

```
$(document).ready(function() {
    $('#selected-plays > li').addClass('horizontal');
});
```

Führen wir den Test jetzt durch, wird er bestanden:

The screenshot shows a web-based testing interface titled 'Appendix B Tests'. It includes a toolbar with a 'Hide passed tests' button. Below it, a status bar indicates 'Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:5.0) Gecko/20100101 Firefox/5.0'. The main area displays test results: 'Tests completed in 19 milliseconds.', '1 tests of 1 passed, 0 failed.', and two specific test cases: '1. Selecting: Child Selector (0, 1, 1)' and '2. Selecting: Attribute Selectors (0, 0, 0)', both of which are marked as 'Rerun'.

Jetzt zeigt der Test *Selecting: Child Selector* keine Fehlschläge und einen bestandenen Test von insgesamt einem Test an. Wir können diesem Test noch einige Attributselektortests hinzufügen:

```
module('Selecting', {
    setup: function() {
        this.topLis = $('#selected-plays > li.horizontal');
    }
});

test('Child Selector', function() {
    expect(1);
    equal(this.topLis.length, 3, 'Top LIs have horizontal class');
});

test('Attribute Selectors', function() {
    expect(2);
    ok(this.topLis.find('.mailto').length == 1, 'a.mailto');
    equal(this.topLis.find('.pdflink').length, 1, 'a.pdflink');
});
```

Hier haben wir einen weiteren Test eingeführt: `ok()`. Dieser nimmt zwei Argumente an: einen Ausdruck, der zu wahr oder falsch führen soll, und eine Beschreibung. Beachten Sie auch, dass wir die lokale Variable `topLis` aus dem *Child Selector*-Test herausgenommen und in die Callback-Funktion des Moduls `setup()` verlegt haben. `module()` akzeptiert ein optionales zweites Argument, eine Map, die eine `setup()`- und eine  `teardown()`-Funktion enthält. Innerhalb dieser Funktionen können wir das Schlüsselwort `this` verwenden, um Variablen mit einem Aufruf allen Modultests zuzuweisen.

Wieder schlagen die neuen Tests ohne den dazugehörigen Code fehl, sodass wir ihn wie folgt in das Skript einfügen:

```
$(document).ready(function() {
    $('#selected-plays > li').addClass('horizontal');
    $('a[href^="mailto:"]').addClass('mailto');
    $('a[href$=".pdf"]').addClass('pdflink');
});
```

Wenn wir den neu erstellten Testsatz aufklappen, sehen wir den Unterschied in der Ausgabe der `ok()`- und `equal()`-Tests wie im folgenden Screenshot:

The screenshot shows the QUnit test results for 'Appendix B Tests'. The browser information is Mozilla/5.0 (Macintosh; Intel Mac OS X 10.6; rv:5.0) Gecko/20100101 Firefox/5.0. The test results indicate that 3 tests passed and 0 failed. The first test, 'Selecting: Child Selector (0, 1, 1)', passed. The second test, 'Selecting: Attribute Selectors (0, 2, 2)', also passed. The third test, 'Expected: 1', is shown as a green box with two items: '1. a.mailto' and '2. a.pdflink'. The text 'Expected: 1' is also present below the list.

Zusätzlich zur grünen Mitteilung, dass beide Tests erfolgreich gelaufen sind, gibt der `equal()`-Test auch das erwartete Ergebnis aus. Da er mehr Informationen liefert, wird der `equal()`-Test normalerweise dem `ok()`-Test vorgezogen.

#### B.4.1 Asynchrones Testen

Asynchrones JavaScript zu testen, wie Ajax-Anfragen, stellt eine besondere Herausforderung dar. Der Rest der Tests muss warten, während der asynchrone Test ausgeführt wird, dann müssen sie erneut ausgeführt werden, wenn der Test abgeschlossen ist. Dieses Szenarium ist uns bereits vertraut: Wir kennen solche asynchronen Operationen schon aus Effektwarteschlangen, Ajax-Callback-Funktionen und Promise-Objekten. In QUnit verwenden wir einen speziellen Testsatz namens `asyncTest()`. Er sieht aus wie ein regulärer `test()`-Satz, nur dass er den Testablauf anhält, bis wir durch einen Aufruf der speziellen `start()`-Funktion zurückkehren:

```
asyncTest('JSON', function() {
    $.getJSON('B.json', function(json, textStatus) {
        // Fügt hier den Test hinzu
    }).always(function() {
        start();
    });
});
```

Hier rufen wir einfach JSON aus B.js ab und lassen die Tests fortfahren, wenn die Anfrage abgeschlossen ist, egal ob sie erfolgreich war oder fehlschlug. In einem richtigen Test würden wir textStatus prüfen, um sicherzustellen, dass die Anfrage erfolgreich war, und den Wert eines der Objekte im JSON-Antwort-Array wie folgt prüfen:

```
asyncTest('JSON', function() {
    expect(2);
    var backbite = {
        "term": "BACKBITE",
        "part": "v.t.",
        "definition": "To speak of a man as you find him when he
can't find you."
    };
    $.getJSON('B.json', function(json, textStatus) {
        equal(textStatus, 'success', 'Request successful');
        deepEqual(json[1], backbite,
            'result array matches "backbite" map');
    }).always(function() {
        start();
    });
});
```

Um den Antwortwert zu testen, können wir eine weitere Testfunktion namens `deepEqual()` verwenden. Wenn normalerweise zwei Objekte verglichen werden, sind sie nicht gleich, wenn sie nicht auf denselben Speicher verweisen. Wenn wir stattdessen die Objektinhalte vergleichen wollen, verwenden wir `deepEqual()`. Diese Funktion geht beide Objekte durch, um sicherzustellen, dass sie die gleichen Eigenschaften mit den gleichen Werten haben.

## B.5 Andere Testarten

QUnit bietet eine ganze Reihe weiterer Testfunktionen. Einige, wie `notEqual()` und `notDeepEqual()`, sind einfach die Umkehrfunktionen der von uns benutzten Funktionen, während andere, wie `strictEqual()` und `raises()`, spezielle Anwendungen haben. Weitere Informationen über diese Funktionen, Definitionen und Beispiele über QUnit im Allgemeinen finden Sie im GitHub-Repository unter <https://github.com/jquery/qunit/>.

## B.6 Praktische Erwägungen

Die Beispiele in diesem Anhang sind zwangsläufig recht einfach. In der Praxis können wir Tests schreiben, die die korrekte Funktion auch komplizierter Mechanismen sicherstellen.

Idealerweise halten wir unsere Tests so kurz und einfach wie möglich, auch wenn die getesteten Mechanismen komplex sind. Indem wir Tests für spezielle Szenarien schreiben, können wir eine gewisse Sicherheit erlangen, dass wir das Gesamtverhalten analysieren, auch wenn wir nicht jede einzelne Möglichkeit prüfen.

Trotzdem ist es möglich, dass in unserem Code Fehler auftreten, obwohl wir dafür Tests geschrieben haben. Wenn Tests erfolgreich laufen und trotzdem Fehler auftreten, ist es nicht der richtige Weg, sofort am Code zu arbeiten, sondern erst einen neuen Test zu schreiben, der das Verhalten überprüft. So stellen wir nicht nur sicher, dass das Problem gelöst ist, sondern auch, dass wir für die Zukunft über einen passenden Test verfügen.

QUnit kann neben dem Testen von Elementen auch zum *funktionalen Testen* eingesetzt werden. Während Unit Tests dazu dienen, die korrekte Funktionsweise von Codeeinheiten (Methoden und Funktionen) zu prüfen, dienen funktionale Tests dazu, die richtige Reaktion auf Benutzereingaben zu testen. In Kapitel 12 haben wir beispielsweise eine Tabellensortierung implementiert. Wir könnten einen Unit Test für die Sortiermethode schreiben, der aufgerufen wird, wenn die Tabelle sortiert wird. Alternativ könnte ein Funktionstest einen Klick des Anwenders auf den Tabellenkopf simulieren und dann überprüfen, ob die Sortierung korrekt erfolgt ist. Spezielle Funktionstest-Frameworks wie Selenium (<http://seleniumhq.org/>) können für solch erweiterte Szenarien eingesetzt werden.

Damit unsere Tests konsistente Ergebnisse liefern, müssen wir an zuverlässigen und konstanten Beispieldaten arbeiten. Wenn wir den jQuery-Code für eine dynamische Seite testen, kann es vorteilhaft sein, eine statische Version der Seite für die Tests zu verwenden. Dieser Ansatz isoliert Ihre Codeelemente und macht es einfacher zu erkennen, welche Fehler vom serverseitigen und welche vom browserseitigen Code herrühren.

### **B.6.1 Literatur**

Diese Überlegungen sind keineswegs erschöpfend. Testgesteuerte Entwicklung ist ein weites Thema und ein kurzer Anhang reicht nicht aus, um es vollständig vorzustellen. Einige Onlinequellen mit weiteren Informationen zu diesem Thema finden Sie unter folgenden URLs:

- Die QUnit-Dokumentation (<http://docs.jquery.com/Qunit>)
- *JavaScript-Testautomatisierung mit QUnit* von Jörn Zaefferer (<http://msdn.microsoft.com/en-us/scriptjunkie/gg749824.aspx>)
- *Testgesteuerte jQuery-Entwicklung* von Elijah Manor (<http://msdn.microsoft.com/en-us/scriptjunkie/ff452703.aspx>)
- *Ansätze für Element-Tests* von Bob McCune (<http://www.bobmccone.com/2006/12/09/unit-testing-best-practices/>)

Es existieren auch viele Bücher zu diesem Thema, wie z.B. Kent Beck, *Test Driven Development: By Example*, und Christian Johansen, *Test-Driven JavaScript Development*.

## **B.7 Zusammenfassung**

Tests mit QUnit zu schreiben kann eine gute Hilfe dabei sein, Ihren jQuery-Code sauber und erweiterbar zu halten. Wir haben einige Wege kennengelernt, wie wir Tests in einem Projekt implementieren, die sicherstellen, dass unser Code wie gewünscht funktioniert. Indem wir kleine, separate Codeelemente testen, vermeiden wir einige der Probleme, die beim Testen komplexer Projekte auftreten. Gleichzeitig können wir effizienter Regressionstests innerhalb eines Projekts durchführen, was uns wertvolle Programmierzeit spart.



# C Kurzreferenz

Dieser Anhang dient als Kurzreferenz für die jQuery-API einschließlich der Selektorausdrücke und Methoden. Eine detaillierte Abhandlung dazu finden Sie im Begleitband, dem *jQuery Reference Guide*, und in der jQuery-Dokumentation unter <http://docs.jquery.com>.

## C.1 Selektorausdrücke

Die jQuery-Factory-Funktion `$()` wird verwendet, um Elemente auf der Seite zu finden, mit denen wir arbeiten wollen. Die Funktion nimmt einen String mit CSS-artiger Syntax entgegen, der Selektorausdruck genannt wird. Selektorausdrücke werden detailliert in Kapitel 2, »*Elemente auswählen*«, besprochen.

### Einfaches CSS

Selektor	Zutreffende Elemente
*	Alle Elemente
#id	Das Element mit der gegebenen ID
element	Alle Elemente des gegebenen Typs
.class	Alle Elemente der gegebenen Klasse
a, b	Elemente, die auf a oder b zutreffen
a b	Elemente b, die Nachfahren von a sind
a>b	Elemente b, die Kinder von a sind
a+b	Elemente b, die direkt auf Geschwister-a-Elemente folgen
a~b	Elemente b, die auf Geschwister-a-Elemente folgen

## Position unter Geschwistern

Selektor	Zutreffende Elemente
:nth-child(index)	Elemente, die ein indexiertes Kind ihres Elternelements sind (1-basiert)
:nth-child(even)	Elemente, die ein gerades Kind ihres Elternelements sind (1-basiert)
:nth-child(odd)	Elemente, die ein ungerades Kind ihres Elternelements sind (1-basiert)
:nth-child(formula)	Elemente, die das n-te Kind ihre Elternelements sind (1-basiert); Formeln der Form $a+b$ gelten für die Ganzzahlen a und b
:first-child	Elemente, die das erste Kind ihres Elternelements sind
:last-child	Elemente, die das letzte Kind ihres Elternelements sind
:only-child	Elemente, die das einzige Kind ihres Elternelements sind

## Position unter passenden Elementen

Selektor	Zutreffende Elemente
:first	Das erste Element im Ergebnissatz
:last	Das letzte Element im Ergebnissatz
:not(a)	Alle Elemente im Ergebnissatz, die nicht auf a zutreffen
:even	Gerade Elemente im Ergebnissatz (0-basiert)
:odd	Ungerade Elemente im Ergebnissatz (0-basiert)
:eq(index)	Ein nummeriertes Element im Ergebnissatz (0-basiert)
:gt(index)	Alle Ergebnisse im Ergebnissatz nach (größer als) dem gegebenen Index (0-basiert)
:lt(index)	Alle Elemente im Ergebnissatz vor (kleiner als) dem gegebenen Index (0-basiert)

## Attribute

Selektor	Zutreffende Elemente
[attr]	Elemente, die das Attribut attr besitzen
[attr="value"]	Elemente, deren attr-Attribut value ist
[attr!="value"]	Elemente, deren attr-Attribut nicht value ist
[attr^="value"]	Elemente, deren attr-Attribut mit value beginnt
[attr\$="value"]	Elemente, deren attr-Attribut mit value endet
[attr*="value"]	Elemente, deren attr-Attribut den Teilstring value enthält
[attr~="value"]	Elemente, deren attr-Attribute eine durch Leerzeichen getrennte Menge von Strings sind, von denen einer value ist
[attr = "value"]	Elemente, deren attr-Attribut entweder gleich value ist oder mit value gefolgt von einem Trennstrich beginnt

## Formulare

Selektor	Zutreffende Elemente
:input	Alle <input>-, <select>-, <textarea>- und <button>-Elemente
:text	<input>-Elemente mit type="text"
:password	<input>-Elemente mit type="password"
:file	<input>-Elemente mit type="file"
:radio	<input>-Elemente mit type="radio"
:checkbox	<input>-Elemente mit type="checkbox"
:submit	<input>-Elemente mit type="submit"
:image	<input>-Elemente mit type="image"
:reset	<input>-Elemente mit type="reset"
:button	<input>-Elemente mit type="button" und <button>-Elemente
:enabled	Aktivierte Formularelemente
:disabled	Deaktivierte Formularelemente
:checked	Markierte Checkboxen und Radio-Schaltflächen
:selected	Ausgewählte <option>-Elemente

## Andere benutzerdefinierte Selektoren

Selektor	Zutreffende Elemente
:header	Header-Elemente (z.B. <h1>, <h2>)
:animated	Elemente mit laufender Animation
:contains(text)	Elemente, die den gegebenen Text enthalten
:empty	Elemente ohne Kindknoten
:has(a)	Elemente mit einem Nachfolgeelement, das auf a zutrifft
:parent	Elemente, die Kindknoten haben
:hidden	Elemente, die verborgen sind, entweder durch CSS oder weil für sie <input type="hidden" /> gilt
:visible	Die Umkehrfunktion von :hidden
:focus	Das Element, das den Tastaturfokus besitzt

## C.2 Methoden zum Durchlaufen des DOM

Nach der Erstellung eines jQuery-Objekts mittels `$()` können wir den Ergebnissatz, mit dem wir arbeiten, verändern, indem wir eine dieser DOM-Traversierungsmethoden aufrufen. Diese Methoden werden detailliert in Kapitel 2 beschrieben.

### Filterung

Methode	Inhalt des zurückgegebenen jQuery-Objekts
<code>.filter(selector)</code>	Ausgewählte Elemente, die auf den gegebenen Selektor zutreffen
<code>.filter(callback)</code>	Ausgewählte Elemente, für die die Callback-Funktion <code>true</code> ergibt
<code>.eq(index)</code>	Das ausgewählte Element am gegebenen 0-basierten Index
<code>.first</code>	Das erste ausgewählte Element
<code>.last</code>	Das letzte ausgewählte Element
<code>.slice(start, [end])</code>	Ausgewählte Elemente im gegebenen Bereich 0-basierter Indizes
<code>.not(selector)</code>	Ausgewählte Elemente, die nicht auf den gegebenen Selektor zutreffen
<code>.has(selector)</code>	Ausgewählte Elemente, die einen Nachfolger haben, der auf <code>selector</code> zutrifft

### Nachfahren

Methode	Inhalt des zurückgegebenen jQuery-Objekts
<code>.find(selector)</code>	Nachfolgeelemente, die auf den Selektor zutreffen
<code>.contents()</code>	Kindknoten (einschließlich Textknoten)
<code>.children([selector])</code>	Kindknoten, optional durch einen Selektor gefiltert

### Geschwister

Methode	Inhalt des zurückgegebenen jQuery-Objekts
<code>.next([selector])</code>	Das Geschwisterelement, das direkt auf das ausgewählte Element folgt, optional durch einen Selektor gefiltert
<code>.nextAll([selector])</code>	Alle Geschwisterelemente, die jedem ausgewählten Element folgen, optional durch einen Selektor gefiltert
<code>.nextUntil([selector], [filter])</code>	Alle Geschwisterelemente, die jedem ausgewählten Element folgen, bis zum (nicht einschließlich) ersten Element, das auf Selektor zutrifft, optional mit einem weiteren Selektor gefiltert

Methode	Inhalt des zurückgegebenen jQuery-Objekts
.prev([selector])	Das Geschwisterelement, das direkt jedem ausgewählten Element vorangeht, optional durch einen Selektor gefiltert
.prevAll([selector])	Alle Geschwisterelemente, die jedem ausgewählten Element vorangehen, optional durch einen Selektor gefiltert
.prevUntil([selector], [filter])	Alle Geschwisterelemente, die jedem ausgewählten Element vorangehen, bis zum (nicht einschließlich) ersten Element, das auf Selektor zutrifft, optional durch einen weiteren Selektor gefiltert
.siblings([selector])	Alle Geschwisterelemente, optional durch einen Selektor gefiltert

## Vorfahren

Methode	Inhalt des zurückgegebenen jQuery-Objekts
.parent([selector])	Das Elternelement jedes ausgewählten Elements, optional durch einen Selektor gefiltert
.parents([selector])	Alle Vorfahren, optional durch einen Selektor gefiltert
.parentsUntil([selector], [filter])	Alle Vorfahren jedes ausgewählten Elements, bis zum (nicht einschließlich) ersten Element, auf das selector zutrifft, optional durch einen weiteren Selektor gefiltert
.closest(selector)	Das erste Element, das auf den Selektor zutrifft, angefangen beim ausgewählten Element und hoch bis zu seinen Vorfahren im DOM-Baum
.offsetParent()	Das positionierte Elternelement (z.B. relativ, absolut) des ersten ausgewählten Elements

## Manipulation von Sammlungen

Methode	Inhalt des zurückgegebenen jQuery-Objekts
.add(selector)	Ausgewählte Elemente und alle zusätzlichen Elemente, die auf den gegebenen Selektor zutreffen
.andSelf()	Die ausgewählten Elemente und zusätzlich der bereits vorhandene Ergebnissatz ausgewählter Elemente auf dem internen jQuery-Stack
.end()	Der vorherige Satz ausgewählter Elemente auf dem internen jQuery-Stack
.map(callback)	Das Ergebnis der Callback-Funktion, die auf jedem einzelnen ausgewählten Element aufgerufen wurde
.pushStack(elements)	Die angegebenen Elemente

## Mit ausgewählten Elementen arbeiten

Methode	Beschreibung
.is(selector)	Entscheidet, ob eines der passenden Elemente auf den gegebenen Selektorausdruck zutrifft
.index()	Holt den Index des passenden Elements in Relation zu seinen Geschwisterelementen
.index(element)	Holt den Index des gegebenen DOM-Knotens innerhalb eines Ergebnissatzes
\$.contains(a,b)	Entscheidet, ob DOM-Knoten b DOM-Knoten a enthält
.each(callback)	Iteriert über die passenden Elemente und führt für jedes ein Callback aus
.length	Holt die Anzahl der passenden Elemente
.get()	Holt ein Array mit DOM-Knoten, die zu den gefundenen Elementen passen
.get(index)	Holt den DOM-Knoten, der zum gefundenen Element am gegebenen Index passt
.toArray()	Holt ein Array mit DOM-Knoten, die zu den gefundenen Elementen passen

## C.3 Ereignismethoden

Um auf den Anwender reagieren zu können, müssen wir unsere Handler mit den Ereignismethoden registrieren. Beachten Sie, dass viele DOM-Ereignisse nur auf bestimmte Elementtypen zutreffen. Diese Feinheiten werden hier jedoch nicht besprochen. Ereignismethoden werden in Kapitel 3 detailliert besprochen.

### Binden

Ereignismethode	Beschreibung
.ready(handler)	Bindet den aufzurufenden Handler, wenn DOM und CSS komplett geladen sind
.bind(type, [data], handler)	Bindet den aufzurufenden Handler, wenn der gegebene Ereignistyp an das Element gesendet wird
.one(type, [data], handler)	Bindet den aufzurufenden Handler, wenn der gegebene Ereignistyp an das Element gesendet wird, entfernt die Bindung, wenn der Handler aufgerufen wird
.unbind([type], [handler])	Entfernt die Bindung eines Elements (für einen Ereignistyp, einen bestimmten Handler oder alle Bindungen)
.live(type, handler)	Bindet den aufzurufenden Handler, wenn der gegebene Ereignistyp mittels Ereignisdelegation an das Element gesendet wird
.die(type, [handler])	Entfernt die Bindung des vorher mit .live() gebundenen Elements

Ereignismethode	Beschreibung
.delegate(selector, type,[data], handler)	Bindet den aufzurufenden Handler, wenn der gegebene Ereignistyp an das nachfolgende Element, das auf selector zutrifft, gesendet wird
.delegate(selector, handlers)	Bindet eine Map von Handlern für Aufruf, wenn die gegebenen Ereignistypen an ein nachfolgendes Element gesendet werden, das auf selector passt
.undelegate(selector, type,[handler])	Entfernt die Bindungen eines vorher mit .delegate() gebundenen Elements

### Binden in Kurzform

Ereignismethode	Beschreibung
.blur(handler)	Bindet den aufzurufenden Handler, wenn das Element den Tastaturfokus verliert
.change(handler)	Bindet den aufzurufenden Handler, wenn sich der Elementwert ändert
.click(handler)	Bindet den aufzurufenden Handler, wenn das Element angeklickt wird
.dblclick(handler)	Bindet den aufzurufenden Handler, wenn das Element doppelt angeklickt wird
.error(handler)	Bindet den aufzurufenden Handler, wenn das Element ein Fehlerereignis empfängt (browserspezifisch)
.focus(handler)	Bindet den aufzurufenden Handler, wenn das Element den Tastaturfokus erhält
.focusin(handler)	Bindet den aufzurufenden Handler, wenn das Element oder ein Nachfahre den Tastaturfokus erhält
.focusout(handler)	Bindet den aufzurufenden Handler, wenn das Element oder ein Nachfahre den Tastaturfokus verliert
.keydown(handler)	Bindet den aufzurufenden Handler, wenn eine Taste gedrückt wird und das Element den Tastaturfokus besitzt
.keypress(handler)	Bindet den aufzurufenden Handler, wenn eine Taste heruntergedrückt wird und das Element den Tastaturfokus besitzt
.keyup(handler)	Bindet den aufzurufenden Handler, wenn die Taste losgelassen wird und das Element den Tastaturfokus besitzt
.load(handler)	Bindet den aufzurufenden Handler, wenn das Element fertig geladen ist
.mousedown(handler)	Bindet den aufzurufenden Handler, wenn die Maustaste innerhalb des Elements gedrückt wird
.mouseenter(handler)	Bindet den aufzurufenden Handler, wenn der Mauszeiger das Element erreicht, keine Auswirkung bei Event Bubbling
.mouseleave(handler)	Bindet den aufzurufenden Handler, wenn der Mauszeiger das Element verlässt, keine Auswirkung bei Event Bubbling

356

Ereignismethode	Beschreibung
.mousemove(handler)	Bindet den aufzurufenden Handler, wenn sich der Mauszeiger innerhalb des Elements bewegt
.mouseout(handler)	Bindet den aufzurufenden Handler, wenn der Mauszeiger das Element verlässt
.mouseover(handler)	Bindet den aufzurufenden Handler, wenn der Mauszeiger das Element erreicht
.mouseup(handler)	Bindet den aufzurufenden Handler, wenn die Maustaste innerhalb des Elements losgelassen wird
.resize(handler)	Bindet den aufzurufenden Handler, wenn das Element vergrößert oder verkleinert wird
.scroll(handler)	Bindet den aufzurufenden Handler, wenn sich die Scrollposition des Elements verändert
.select(handler)	Bindet den aufzurufenden Handler, wenn Text im Element ausgewählt wird
.submit(handler)	Bindet den aufzurufenden Handler, wenn das Formularelement übertragen wird
.unload(handler)	Bindet den aufzurufenden Handler, wenn das Element aus dem Speicher entfernt wird

### Spezielle Abkürzungen

Ereignismethode	Beschreibung
.hover(enter, leave)	Bindet enter für Aufruf, wenn die Maus ein Element erreicht, und leave, wenn sie es verlässt
.toggle(handler1, handler2, ...)	Bindet handler1 für Aufruf, wenn die Maus auf dem Element geklickt wird, gefolgt von handler 2 usw. für die weiteren Klicks.

### Auslösungen (Trigger)

Ereignismethode	Beschreibung
.trigger(type, [data])	Löst Handler für das Ereignis auf dem Element aus und führt dafür eine Standardaktion aus
.triggerHandler(type, [data])	Löst Handler für das Ereignis auf dem Element aus, ohne eine Standardaktion auszuführen

### Abkürzungen für Trigger

Ereignismethode	Beschreibung
.blur()	Löst das blur-Ereignis aus
.change()	Löst das change-Ereignis aus
.click()	Löst das click-Ereignis aus

Ereignismethode	Beschreibung
.dblclick()	Löst das dblclick-Ereignis aus
.error()	Löst das error-Ereignis aus
.focus()	Löst das focus-Ereignis aus
.keydown()	Löst das keydown-Ereignis aus
.keypress()	Löst das keypress-Ereignis aus
.keyup()	Löst das keyup-Ereignis aus
.select()	Löst das select-Ereignis aus
.submit()	Löst das submit-Ereignis aus

## Utility

Ereignismethode	Beschreibung
\$.proxy(fn, context)	Erzeugt eine neue Funktion, die den gegebenen Kontext ausführt

## C.4 Effektmethoden

Diese Effektmethoden können verwendet werden, um auf DOM-Elementen Animationen auszuführen. Effektmethoden werden detailliert in Kapitel 4 besprochen.

### Vordefinierte Effekte

Effektmethode	Beschreibung
.show()	Zeigt die passenden Elemente an
.hide()	Verbirgt die passenden Elemente
.show(speed, [callback])	Zeigt die passenden Elemente durch Animation von height, width und opacity an
.hide(speed, [callback])	Verbirgt die passenden Elemente durch Animation von height, width und opacity
.toggle([speed], [callback])	Zeigt oder verbirgt die passenden Elemente
.slideDown([speed], [callback])	Zeigt die passenden Elemente mit einer gleitenden Bewegung an
.slideUp([speed], [callback])	Verbirgt die passenden Elemente mit einer gleitenden Bewegung
.slideToggle([speed], [callback])	Zeigt oder verbirgt die passenden Elemente mit einer gleitenden Bewegung
.fadeIn([speed], [callback])	Zeigt die passenden Elemente durch Deckkrafterhöhung an

Effektmethode	Beschreibung
.fadeOut([speed], [callback])	Verbirgt die passenden Elemente durch Deckkraftverringerung
.fadeToggle([speed], [callback])	Zeigt oder verbirgt die passenden Elemente durch eine Einblendanimation
.fadeTo(speed, opacity, [callback])	Passt die Deckkraft der passenden Elemente an

## Benutzerdefinierte Animationen

Effektmethode	Beschreibung
.animate(attributes, [speed], [easing], [callback])	Führt eine benutzerdefinierte Animation für die angegebenen CSS-Attribute aus
.animate(attributes, options)	Eine Low-Level-Schnittstelle für .animate() mit Zugang zur Animationswarteschlange

## Warteschlangenmanipulation

Effektmethode	Beschreibung
.queue([queueName])	Ruft die Warteschlange der Funktionen für das erste passende Element ab
.queue([queueName], callback)	Fügt am Ende der Warteschlange callback hinzu
.queue([queueName], newQueue)	Ersetzt die Warteschlange durch eine neue
.dequeue([queueName])	Führt die nächste Funktion in der Warteschlange aus
.clearQueue([queueName])	Entfernt alle ausstehenden Funktionen aus der Warteschlange
.stop([clearQueue], [jumpToEnd])	Hält die aktuelle Animation an und beginnt die Animationen in der Warteschlange, wenn vorhanden
.delay(duration, [queueName])	Wartet duration in Millisekunden, bevor das nächste Element der Warteschlange ausgeführt wird
.promise([queueName], [target])	Gibt ein Promise-Objekt zurück, wenn alle Aktionen in der Warteschlange ausgeführt sind

## C.5 DOM-Manipulationsmethoden

DOM-Manipulationsmethoden werden detailliert in Kapitel 5 beschrieben.

### Attribute und Eigenschaften

Manipulationsmethode	Beschreibung
.attr(key)	Holt das Attribut namens key
.attr(key, value)	Setzt das Attribut namens key auf value
.attr(key, fn)	Setzt das Attribut key auf das Ergebnis von fn (für jedes zutreffende Element separat ausgeführt)
.attr(map)	Setzt als Schlüssel-Wert-Paare übergebene Attributwerte
.removeAttr(key)	Entfernt das Attribut namens key
.prop(key)	Holt die Eigenschaft namens key
.prop(key, value)	Setzt die Eigenschaft namens key auf value
.prop(key, fn)	Setzt die Eigenschaft namens key auf das Ergebnis von fn (für jedes zutreffende Element separat aufgerufen)
.prop(map)	Setzt Eigenschaften, die als Schlüssel-Wert-Paare übergeben werden
.removeProp(key)	Entfernt die Eigenschaft namens key
.addClass(class)	Fügt jedem passenden Element die gegebene Klasse hinzu
.removeClass(class)	Entfernt die gegebene Klasse von jedem passenden Element
.toggleClass(class)	Entfernt die gegebene Klasse, wenn vorhanden, und fügt ansonsten die Klasse jedem passenden Element hinzu
.hasClass(class)	Gibt true zurück, wenn eines der passenden Elemente die gegebene Klasse besitzt
.val()	Holt den Wert des ersten passenden Elements
.val(value)	Setzt das Wertattribut jedes Elements auf value

### Inhalt

Manipulationsmethode	Beschreibung
.html()	Holt den HTML-Inhalt des ersten passenden Elements
.html(value)	Setzt den HTML-Inhalt des ersten passenden Elements auf value
.text()	Holt den Textinhalt aller passenden Elemente als einzelnen String
.text(value)	Setzt den Textinhalt jedes passenden Elements auf value

## CSS

Manipulationsmethode	Beschreibung
.css(key)	Holt das CSS-Attribut namens key
.css(key, value)	Setzt das CSS-Attribut namens key auf value
.css(map)	Setzt als Schlüssel-Wert-Paare übergebene Attributwerte

## Abmessungen

Manipulationsmethode	Beschreibung
.offset()	Holt die oberen linken Pixelkoordinaten des ersten passenden Elements, relativ zum Ansichtsfenster
.position()	Holt die oberen linken Pixelkoordinaten des ersten passenden Elements relativ zum durch .OffsetParent() zurückgegebenen Element
.scrollTop()	Holt die vertikale Scrollposition des ersten passenden Elements
.scrollTop(value)	Setzt die vertikale Scrollposition aller passenden Elemente auf value
.scrollLeft()	Holt die horizontale Scrollposition des ersten passenden Elements
.scrollLeft(value)	Setzt die horizontale Scrollposition aller passenden Elemente auf value
.height()	Holt die Höhe des ersten passenden Elements
.height(value)	Setzt die Höhe der passenden Elemente auf value
.width()	Holt die Breite des ersten passenden Elements
.width(value)	Setzt die Breite des ersten passenden Elements auf value
.innerHeight()	Holt die Höhe des ersten passenden Elements einschließlich Innenrand, aber ohne Rahmen
.innerWidth()	Holt die Breite des ersten passenden Elements einschließlich Innenrand, aber ohne Rahmen
.outerHeight(includeMargin)	Holt die Höhe des ersten passenden Elements einschließlich Innenrand, Rahmen und optionalem Außenrand
.outerWidth(includeMargin)	Holt die Breite des ersten passenden Elements einschließlich Innenrand, Rahmen und optionalem Außenrand

## Einfügen

Manipulationsmethode	Beschreibung
.append(content)	Fügt content am Ende jedes passenden Elements an
.appendTo(selector)	Fügt die passenden Elemente am Ende der zu selector passenden Elemente an
.prepend(content)	Fügt content am Anfang jedes passenden Elements an
.prependTo(selector)	Fügt die passenden Elemente am Anfang der zu selector passenden Elemente an
.after(content)	Fügt content nach jedem passenden Element ein
.insertAfter(selector)	Fügt die passenden Elemente nach jedem zu selector passenden Element ein
.before(content)	Fügt content vor jedem passenden Element ein
.insertBefore(content)	Fügt die passenden Elemente vor jedem zu selector passenden Element ein
.wrap(content)	Verpackt jedes passende Element innerhalb von content
.wrapAll(content)	Verpackt alle passenden Elemente als Einheit innerhalb von content
.wrapInner(content)	Verpackt den inneren Inhalt jedes passenden Elements in content

## Ersetzen

Manipulationsmethode	Beschreibung
.replaceWith(content)	Ersetzt die passenden Elemente durch content
.replaceAll(selector)	Ersetzt alle zum selector passenden Elemente durch die passenden Elemente

## Entfernen

Manipulationsmethode	Beschreibung
.empty()	Entfernt die Kindknoten aus jedem passenden Element
.remove([selector])	Entfernt die passenden Knoten (optional durch selector gefiltert) aus dem DOM
.detach([selector])	Entfernt die passenden Knoten (optional durch selector gefiltert) aus dem DOM, behält jedoch angefügte jQuery-Daten
.unwrap()	Entfernt das Elternelement des Elements

## Kopieren

Manipulationsmethode	Beschreibung
.clone([withHandlers])	Erstellt eine Kopie aller passenden Elemente und kopiert optional auch die Ereignishandler

## Daten

Manipulationsmethode	Beschreibung
.data(key)	Holt ein Datenobjekt namens key, das zum ersten passenden Element gehört
.data(key, value)	Setzt ein Datenobjekt namens key, das zu jedem passenden Element gehört, auf value
.removeData(key)	Entfernt das Datenobjekt namens key, das zu jedem passenden Element gehört

## C.6 Ajax-Methoden

Wir können vom Server Informationen ohne ein Neuladen der Seite erhalten, indem wir eine der folgenden Ajax-Methoden verwenden. Ajax-Methoden werden detailliert in Kapitel 6 besprochen.

### Anfragen senden

Ajax-Methode	Beschreibung
\$.ajax(options)	Erstellt eine Ajax-Anfrage mit den angegebenen Optionen; es handelt sich um eine Low-Level-Methode, die normalerweise durch übergeordnete Methoden aufgerufen wird
.load(url, [data], [callback])	Startet eine Ajax-Anfrage an url und platziert die Antwort in die passenden Elemente
\$.get(url, [data], [callback], [returnType])	Startet mit der GET-Methode eine Anfrage an url
\$.getJSON(url, [data], [callback])	Startet eine Ajax-Anfrage an url und interpretiert die Antwort als JSON-Datenstruktur
\$.getScript(url, [callback])	Startet eine Anfrage an url und führt die Antwort als JavaScript aus
\$.post(url, [data], [callback], [returnType])	Startet mit der POST-Methode eine Ajax-Anfrage an url

## Anfrageüberwachung

Ajax-Methode	Beschreibung
.ajaxComplete(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion abgeschlossen wird
.ajaxError(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion mit Fehler abgeschlossen wird
.ajaxSend(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion begonnen wird
.ajaxStart(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion begonnen wird und keine anderen aktiv sind
.ajaxStop(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion endet und keine anderen mehr aktiv sind
.ajaxSuccess(handler)	Bindet den aufzurufenden Handler, wenn eine Ajax-Transaktion erfolgreich abgeschlossen wird

## Konfiguration

Ajax-Methode	Beschreibung
\$.ajaxSetup(options)	Setzt Vorgabeoptionen für alle folgenden Ajax-Transaktionen
\$.ajaxPrefilter([dataTypes], handler)	Modifiziert die Optionen für jede Ajax-Anfrage, bevor Sie von \$.ajax() bearbeitet wird
\$.ajaxTransport(transport-Function)	Definiert einen neuen Transportmechanismus für Ajax-Transaktionen

## Hilfsfunktionen

Ajax-Methode	Beschreibung
.serialize()	Codiert die Werte eines Formularelementesatzes in einen Abfragestring
.serializeArray()	Codiert die Werte eines Formularelementesatzes in eine JavaScript-Datenstruktur
\$.param(map)	Codiert eine beliebige Map mit Werten in einen Abfragestring
\$.globalEval(code)	Wertet den gegebenen JavaScript-String im globalen Kontext aus
\$.parseJSON(json)	Konvertiert den gegebenen JSON-String in ein JavaScript-Objekt
\$.parseXML(xml)	Konvertiert den gegebenen XML-String in ein XML-Dokument

## C.7 Verzögerte Objekte

Verzögerte Objekte und deren Promises ermöglichen es uns, auf den Abschluss lange laufender Aufgaben mit einer praktischen Syntax zu reagieren. Sie werden detailliert in Kapitel 11 beschrieben.

### Objekterstellung

Funktion	Beschreibung
<code>\$.Deferred([setupFunction])</code>	Gibt ein neues verzögertes Objekt zurück
<code>\$.when(deferreds)</code>	Gibt ein Promise-Objekt zurück, das aufgelöst wird, wenn die gegebenen verzögerten Objekt aufgelöst werden

### Methoden verzögerter Objekte

Methode	Beschreibung
<code>.resolve([args])</code>	Setzt den Status für das aufzulösende Objekt
<code>.resolveWith(context, [args])</code>	Setzt den Status für das aufzulösende Objekt und bezieht das Schlüsselwort <code>this</code> innerhalb von Callbacks auf <code>context</code>
<code>.reject([args])</code>	Setzt den Status für das aufzulösende Objekt
<code>.rejectWith(context, [args])</code>	Setzt den Status für das aufzulösende Objekt und bezieht das Schlüsselwort <code>this</code> innerhalb von Callbacks auf <code>context</code>
<code>.promise([target])</code>	Gibt ein dem verzögerten Objekt entsprechendes Promise-Objekt zurück

### Methoden für Promise-Objekte

Methode	Beschreibung
<code>.done(callback)</code>	Führt <code>callback</code> aus, wenn das Objekt aufgelöst wird
<code>.fail(callback)</code>	Führt <code>callback</code> aus, wenn das Objekt zurückgewiesen wird
<code>.always([callback])</code>	Führt <code>callback</code> aus, wenn das Objekt aufgelöst oder zurückgewiesen wird
<code>.then(doneCallbacks, failCallbacks)</code>	Führt <code>doneCallbacks</code> aus, wenn das Objekt aufgelöst wird, und <code>failCallbacks</code> , wenn es zurückgewiesen wird
<code>.isRejected()</code>	Gibt <code>true</code> zurück, wenn das Objekt zurückgewiesen wird
<code>.isResolved()</code>	Gibt <code>true</code> zurück, wenn das Objekt aufgelöst wird
<code>.pipe([doneFilter], [failFilter])</code>	Gibt ein neues Promise-Objekt zurück, das aufgelöst wird, wenn das Original-Promise aufgelöst wird, optional nach einer Filterung des Objektstatus mit einer Funktion

## C.8 Verschiedene Eigenschaften und Funktionen

Diese Hilfsmethoden passen in keine der vorangegangenen Kategorien, sind aber oft sehr nützlich, wenn Sie mit jQuery Skripte schreiben.

### Eigenschaften des jQuery-Objekts

Eigenschaft	Beschreibung
<code>\$.support</code>	Gibt eine Map mit Eigenschaften zurück, die anzeigen, welche Funktionen und Standards der Browser unterstützt

### Arrays und Objekte

Funktion	Beschreibung
<code>\$.each(collection, callback)</code>	Iteriert über collection und führt callback für jedes Element aus
<code>\$.extend(target, addition,...)</code>	Modifiziert das Objekt target durch Hinzufügen von Eigenschaften aus anderen Objekten
<code>\$.grep(array, callback, [invert])</code>	Filtert array mit callback als Text
<code>\$.makeArray(object)</code>	Konvertiert object in ein Array
<code>\$.map(array, callback)</code>	Erzeugt ein neues Array, das aus dem Ergebnis von callback besteht, das für jedes Element aufgerufen wird
<code>\$.inArray(value, array)</code>	Bestimmt die Position eines Werts im Array oder gibt -1 zurück, wenn der Wert nicht im Array enthalten ist
<code>\$.merge(array1, array2)</code>	Führt den Inhalt von array1 und array2 zusammen
<code>\$.unique(array)</code>	Entfernt alle doppelten DOM-Elemente aus array

### Objekt-Untersuchungen

Funktion	Beschreibung
<code>\$.isArray(object)</code>	Bestimmt, ob object ein echtes JavaScript-Array ist
<code>\$.isEmptyObject(object)</code>	Bestimmt, ob object leer ist
<code>\$.isFunction(object)</code>	Bestimmt, ob object eine Funktion ist
<code>\$.isPlainObject(object)</code>	Bestimmt, ob object als Literal oder mit new Object erstellt wurde
<code>\$.isWindow(object)</code>	Bestimmt, ob object ein Browserfenster darstellt
<code>\$.isXMLDoc(object)</code>	Bestimmt, ob object ein XML-Knoten ist
<code>\$.type(object)</code>	Holt die JavaScript-Klasse von object

## Sonstige

Funktion	Beschreibung
<code>\$.trim(string)</code>	Entfernt den Leerraum von den Enden eines Strings
<code>\$.noConflict([removeAll])</code>	Wandelt alle \$ in die Pre-jQuery-Definition um
<code>\$.noop()</code>	Eine Funktion ohne Funktion
<code>\$.now()</code>	Die aktuelle Zeit in Sekunden seit dem 1. Januar 1970

# Index

## A

a b 349  
Absolute Relative 98  
Abstraktionsschicht 10  
add(selector) 353  
AHAH 137  
Ajax  
Anforderungen drosseln 312  
Anfragemethoden 362  
Daten bei Bedarf laden 135  
Datenformat 149  
Datentypkonverter 313  
Einführung 9, 135  
Ereignisse 164  
Erweitern 313  
Erweitertes Ajax 301  
Hilfsfunktionen 363  
HTML anhängen 137  
Konfigurationsmethoden 363  
Prefilter 318  
Promise 309  
Sicherheitseinschränkungen 165  
Transport 318  
Überwachungsmethoden 363  
Umgebung 162  
Ajax-Anforderungen  
Beobachterfunktionen 160  
Unterschiedliche Inhalt liefern 158  
Ajax-Methoden 362  
Anfragen senden 362  
Anfrageüberwachung 363  
Hilfsfunktionen 363  
Konfiguration 363  
Ajax-Werkzeugkasten  
Grundlegende Methoden 168  
HTML-Seite laden 170  
Standardoptionen ändern 169  
Zusätzliche Optionen 168

alert() 323  
Allmähliche Funktionsverminderung 26, 76, 290  
Aneinander gereihte Effekte 99  
Animation  
Anhalten 263  
Animationsstatus 262  
Beobachten 261  
Effektmethoden 358  
Einführung 81  
Unterbrechen 261  
Animations-Promise 270  
Anonyme Funktion 16, 143, 329, 333  
Apache 151  
API (Application Programming Interface) 8  
Arrays  
Eigenschaften 365  
Literale 141  
Schreibweise 285  
Sortierung 282  
Asynchrone Aufrufe 139  
Asynchrones JavaScript 344  
Asynchronous HTTP and HTML 137  
Attribute  
Bearbeiten 109  
Nicht-Klassenattribute 110  
Attributselektoren 350  
Einführung 29  
Links formatieren 29  
Aufrufen von Funktionen 47  
a+b 349  
a, b 349  
a>b 349  
a~b 349

## B

Benutzerdefinierte Animationen  
    Effekte 94  
    Erstellen 93, 94  
    Mehrere Eigenschaften 95  
    Positionieren 98  
Benutzerdefinierte Ereignisse  
    Benutzerdefinierte  
        Ereignisparameter 252  
    Einführung 249  
    Parameter 252  
    Unendliches Scrollen 251  
Benutzerinteraktion simulieren 75  
Beobachterfunktionen 160  
Besondere Ereignisse 255  
blur 57  
<body> 24, 340  
Boolesche Attribute 114  
Bubblesort 282  
buildItem() 305  
buildRows() 290, 292, 294  
buildRow() 290  
build\_entry() 156

## C

C 323  
Callbackfunktion 206  
    Einführung 105, 142  
    Verwenden 105  
Callbacks 140, 303  
callback() 320  
CDNs 11  
cellIndex 236  
change 57  
checked 114  
checkScrollPosition() 251  
Chrome 24  
Chrome Developer Tools 18  
class 109  
click 57, 60  
Closing-Umgebung 336  
Closures  
    Einführung 323  
    Ereignishandler 330  
    Interaktion 328

## Closures (*Fortsetzung*)

    jQuery 329  
    Verweisschleifen 335  
    \$(document).ready() 329  
Code verbergen 324  
Codeausführung 45, 46  
Codeoptimierung 55  
complete() 320  
console.log() 21  
contents 313  
converters 314  
copyOffset() 207  
Cross-Site Scripting 165  
CSS 7, 8, 23  
    Attributselektoren 29  
    Elementselektoren 349  
    Inline-Bearbeitung 81  
    Listenelemente formatieren 27  
    Manipulationsmethoden 360  
    Pseudoklassen 32  
    Regeln 136  
    Selektoren 9, 23, 26, 349  
    Selektor-Engine 219

## Cycle

    Download 176  
    Einführung 176  
    Parameter 179  
    Plug-in-Methodenparameter 178  
    Verweise 176  
    Verwenden 176

## D

data-book 292  
data-sort 289  
data\* 289  
Daten  
    An den Server übergeben 150  
    Bei Bedarf laden 135  
Datenbank 275  
Datenbearbeitungsmethoden 362  
Datenformat 149  
Datenspeicherung  
    Einführung 282  
    Nicht-String-Daten 284  
    Sortierrichtung 286  
    Zusätzliche Vorberechnungen 283

- dblclick 57
- deepEqual() 345
- <div> 24, 136
- DOM
  - Abkürzungen für Trigger 356
  - Abmessungsbearbeitung 360
  - Attributbearbeitung 109, 359
  - Baum 46
  - Benutzerdefinierte Animation 358
  - Binden in Kurzform 355
  - CSS-Bearbeitung 360
  - Datenbearbeitung 362
  - Durchquerungsmethoden 37
  - Eigenschaften 113
  - Eigenschaftenbearbeitung 359
  - Einfügen 361
  - Einführung 7, 23, 109
  - Entfernen 361
  - Ereignisse binden 354
  - Ersetzen 361
  - Familienstammbaum 23
  - Hilfsmethoden 357
  - Inhaltsbearbeitung 359
  - Kopieren 362
  - Kurzformen für Ereignisse 57
  - Manipulationsmethoden 359
  - Spezielle Abkürzungen 356
  - Trigger 356
  - Vordefinierte Effekte 357
  - Warteschlangenmanipulation 358
- DOM-Baum bearbeiten
  - Einführung 114
  - Elemente kopieren 125
  - Elemente verschachteln 119
  - Elemente verschieben 117
  - Expliziter Iterator 119
  - Neue Elemente einfügen 116
  - Neue Elemente erstellen 115
  - Umgekehrte Einfügemethoden 121
  - \$() 114
- DOM-Bearbeitungsmethoden 132
- DOM-Durchquerungsmethoden
  - Einführung 37
  - Einzelne Zellen formatieren 38
  - Verkettungsfähigkeiten 40
- DOM-Elemente
  - Eigenschaften 113
  - Stack 234
  - Zugriff 41
- DOM-Manipulationsmethoden
  - Abmessungen 360
  - Attribute und Eigenschaften 359
  - CSS 360
  - Daten 362
  - Einfügen 361
  - Entfernen 361
  - Ersetzen 361
  - Inhalt 359
- DOM-Methoden 229
- DOM-Traversierungsmethoden
  - Ausgewählte Elemente 354
  - DOM-Elementstack 234
  - Einführung 231
  - Filtermethoden 352
  - Geschwistermethoden 352
  - jQuery-Objekteigenschaften 232
  - Nachfahrenmethoden 352
  - Performance 237
  - Plug-in 235
  - Sammlungsbearbeitung 353
  - Vorfahrenmethoden 353
- Download
  - jQuery 10
  - QUnit 339
- Durchlaufen des DOM, Methoden
  - Filterung 352
  - Geschwister 352
  - Manipulation von Sammlungen 353
  - Mit ausgewählten Elementen arbeiten 354
  - Nachfahren 352
  - Vorfahren 353
- Dynamische Tabellenfilterung 221
- E**
  - easeInExpo 184
  - easeInQuart 268
  - Easing 184
  - Effekte
    - Aneinandergereiht 99
    - Animation 259
    - Einführung 81, 259
    - Gleichzeitig 99
    - Globale Eigenschaften 264
    - Mehrere Eigenschaften 268
    - Verzögerte Objekte 268

- Effektmethoden 357
  - Benutzerdefinierte Animationen 358
  - Vordefinierte Effekte 357
  - Warteschlangenmanipulation 358
- Effektmodul
  - Einführung 183
  - Erweitertes Easing 184
  - Farbanimationen 183
  - Klassenanimationen 183
  - Zusätzliche Effekte 185
- Einfügemethoden 361
- element 227, 349
- Elemente
  - DOM-Elemente sortieren 280
  - Einfügen 116
  - Erstellen 115
  - JavaScript-Arrays 279
  - Kopieren 125
  - Verpacken 119
  - Verschieben 117
- Entprellen 255
- Entkoppelt 249
- Entwicklungswerkzeuge
  - Chrome Developer Tools 18
  - Einführung 18
  - Firebug 18
  - Funktionen 18
  - Internet Explorer-
    - Entwicklertools 18
  - Safari Web Inspector 18
- Ereignisdelegierung
  - Delegationsmethode auswählen 247
  - Delegationsmethode verwenden 247
  - Einführung 68, 245
  - Frühe Delegation 248
  - Implementieren 245, 247
  - Kontextargument 249
  - Methoden 70, 71
- Ereignishandler
  - Einführung 330
  - Entfernen 71
  - Erneut binden 73
  - Kontext 53
- Ereignismethoden
  - Abkürzungen für Trigger 356
  - Auslösungen (Trigger) 356
  - Binden 354
  - Binden in Kurzform 355
  - Spezielle Abkürzungen 356
  - Utility 357
- Ereignisobjekt
  - Einführung 64
  - Verwendung 66
  - Ziel 66
  - Zweck 64
- Ereignisse
  - Ajax 164
  - Auslösen 356
  - Benutzerdefinierte Ereignisse 249
  - Benutzerinteraktion simulieren 75
  - Besondere Ereignisse 255
  - Binden 354
  - Binden in Kurzform 355
  - Daten anzeigen 244
  - Drosseln 253
  - Einführung 24, 241
  - Enprellen 255
  - Ereignisdelegierung 68, 245
  - Ereigniskontext 56
  - Ereignismethoden 354
  - Ereignisse abfangen 62
  - Erneut binden 73
  - Formatwechsler 49
  - keyup 312
  - Kurzformen 57
  - Mausereignisse 76
  - Namensräume 72
  - Schaltflächen 52, 53
  - Standardaktionen 67
  - Tastaturereignisse 76
  - throttledScroll 312
  - Trigger 356
  - Zusätzliche Datenseiten laden 243, 244
- Ereignisweiterleitung
  - Abbrechen 66
  - Einführung 62
- error 57
- Ersetzungsmethoden 361
- Erweiterte Attribute
  - Abgekürzte Elementerstellung 295
  - Hooks 296
- Erweiterte DOM-Bearbeitung
  - Daten sortieren 282
  - Elemente verschieben 278
  - Tabellenzeilen 275
  - Zeilen erstellen 290
- Erweiterte Easingfunktionen 184

**E** Erweitertes Ajax

Ajax-Anforderungen drosseln 312  
Ajax-Fähigkeiten erweitern 313  
Fehlerbehandlung 306  
Fortschreitende Verbesserung 301  
jqXHR 308  
eval() 165  
Event Bubbling 63, 64  
event.target 66  
expando 283  
expect() 342  
Explizite Iteration  
    Einführung 25  
    Verwendung 119, 122  
Expliziter Iterator 119

**F**

Familienstammbaum 23  
Fehlerbehandlung 162, 164  
filter 297  
Filter und Streifenmuster 225  
Filterfunktion 37  
Filtermethoden 352  
Firebug 18, 19, 24  
Firefox 24  
FireQuery 19  
Flatland 117  
focus 57  
Formatwechsler 49  
Formular 156  
Formularselektoren 351  
Fortschreitende Verbesserung 26, 49, 61,  
    136, 159, 277, 301  
Fragment 138  
Freie Variablen 328  
function 73  
Funktionale Tests 346  
Funktionen  
    Deklaration 73, 143  
    Verweise 74  
    Zum Namensraum hinzufügen 194  
    Zuweisen 46

**G**

Garbage Collection 326, 334  
Geschwistermethoden 352  
GET 151, 298  
Get-Methoden 128  
GitHub 175, 339  
Gleichzeitige Effekte 99  
Globale Effekteigenschaften  
    Animationen glätten 265  
    Effekt deaktivieren 264  
    Effektdauer 265  
    Einführung 264  
Globale Funktionen  
    Hinzufügen 194  
    jQuery-Funktionen 142  
    Plug-ins 180  
Globale Variablen 325  
Globaler Kontext 145  
Globales jQuery-Objekt 142  
globalVar() 325  
GNU Public License 10  
goToEnd 263  
Grundlegende Methoden 168  
Gültigkeitsbereich von Variablen 326

**H**

Handler  
    Binden 331  
    click 332  
    <head> 24, 340  
Hooks  
    Einführung 296  
    Finden 298  
    Schreiben 297  
    \$.attrHooks 296  
    \$.cssHooks 297  
    \$.propHooks 296  
    \$.valHooks 296  
href 243  
HTML  
    Anhängen 137  
    Attribute 113  
    Benutzerdefinierte Datenattribute in  
        HTML5 288

- HTML (*Fortsetzung*)**
- Einführung 7
  - Fragmente 149
  - HTML-Seite laden 170
  - jQuery einrichten 11
  - HTML-Testdokument 340
  - <html> 24
- I**
- ID 25
  - id 110
  - <iframe> 166
  - Immediately Invoked Function Expression (IIFE) 193, 249, 332
  - Implizite Iteration 10, 15, 55, 201, 216
  - Index 41
  - index 227
  - Inhalte für Ajax-Anforderungen 158
  - Inhaltsbearbeitungsmethoden 359
  - Innere Funktionen
    - Deklaration 323
    - Gültigkeitsbereich von Variablen 326
  - innerFn() 324
  - Instanzmethoden 329
  - Instanzvariablen 329
  - Interaktionsindex 143
  - Interaktionskomponenten 186
  - Interne Zitate 126
  - Internet Explorer 336
  - Internet Explorer-Entwicklertools 18
- J**
- JavaScript
    - Ausführen 145
    - Benchmarking-Website 230
    - Closures 323, 328, 329
    - Dateien 150
    - Einfache Ereignisse 49
    - Innere Funktionen 323
    - Objekte 140
    - Speicherlecks 334
  - Jokerzeichen 29
  - jQuery
    - Aufgaben 7
    - Ausführen 15
- jQuery (*Fortsetzung*)**
- Bibliothek 7, 26
  - Dokumentation 299
  - Download 11
  - Einführung 7
  - Einrichten 11
  - Entwicklungswerzeuge 18, 19
  - Ereignisse 241
  - Formularselektoren 36
  - Forum 175
  - Im Vergleich mit JavaScript 17
  - jQuery-Code hinzufügen 14
  - jQuery-Selektoren 31
  - Namensraum 142, 194
  - Objekteigenschaften 365
  - Objektinstanz 15
  - Objektmethoden 199
  - Objektmethodenkontext 200
  - Plug-in-Anforderungen 216
  - Plug-in-Architektur 175
  - Plug-ins veröffentlichen 217
  - Strategien 9, 10
  - URL 10
  - \$() 25
- jQuery Plugin Repository 217
- jQuery UI**
- Dokumentation 182
  - Effektmodul 182
  - Einführung 182
  - Interaktionskomponenten 186
  - Interaktive Elemente 187
  - jQuery-UI-ThemeRoller 190
  - Widgets 187
- jQuery UI-Widget-Factory**
- Einführung 209
  - Untermethoden 214
  - Widget-Ereignisse 215
  - Widget-Optionen 213
  - Widgets aktivieren 213
  - Widgets deaktivieren 213
  - Widgets entfernen 212
  - Widgets erstellen 210
- jQuery-Closures**
- Anonyme Funktionen 333
  - Benannte Funktionen 333
  - Einführung 329
  - Ereignishandler 330
  - Handler binden 331
  - \$(document).ready() 329

- jQuery-Effekte  
  Ausblenden 90  
  Auseinanderfalten 91  
  Einblenden 90  
  Einführung 89  
  Geschwindigkeit 89  
  Zusammenfalten 91  
  Zusammengesetzte Effekte 92
- jQuery-Objekt 24  
  Eigenschaften 232  
  Instanz 15, 142  
  \$ 365
- jQuery-Objektmethoden  
  Hinzufügen 199  
  Implizite Iteration 201  
  Methodenverkettung 202
- jQuery-Plug-ins  
  Mehrere Funktionen 196  
  Methodenparameter 203  
  Neue globale Funktionen 194  
  \$ 193
- jQuery-Selektoren  
  Einführung 23, 31  
  Plug-in 180, 226  
  Zeilen abwechselnd formatieren 32  
  :even 32  
  :odd 32
- jQuery-Webseiten  
  Einführung 11  
  Gedichttext 15  
  jQuery-Code hinzufügen 14  
  Neue Klassen 15
- jquery.cycle.js 176
- jQuery.mathUtils 198
- jQuery.noConflict() 48
- jqXHR  
  Ajax-Promise 309  
  Antworten zwischenspeichern 310  
  Einführung 309  
  .abort() 308  
  .responseText 308  
  .responseXML 308  
  .setRequestHeader() 308  
  .status 308  
  .statusText 308
- JSON  
  Abrufen 140  
  Dateien 150  
  JSONP 166, 168
- JSON (Fortsetzung)  
  Objekt bearbeiten 292  
  URL 141  
  Zeilen erstellen 290  
  Zeilen sortieren 290  
  jsperf.com 230
- K**
- keydown 57, 76  
keypress 58, 76  
keyup 58, 76, 312  
Kindkombinator 28  
Klasse 25  
Komparatorfunktion 279  
Kontext 199  
Kopiermethoden 362  
Kurzformen für Ereignisse 57
- L**
- Lambda-Funktion 16  
Links 29  
<link> 340  
Listenelemente 27  
load 58  
loading 310  
Lokale Variable 73  
Löschmethoden 361
- M**
- Maps 204  
Mashups 150  
matches 227  
Mausereignisse 76  
Mehrere Eigenschaften 95  
Mehrere Funktionen 196  
Methodenparameter  
  Anpassbare Voreinstellungen 208  
  Callback-Funktion 206  
  Einführung 203  
  Maps 204  
  Voreinstellungen 205  
Mikrooptimierung 228  
MIME-Typ 147  
MIT-Lizenz 10

module() 341  
mousedown 58  
mouseenter 64, 244  
mouseleave 64, 244  
mousemove 58  
mouseout 58, 64  
mouseover 58  
mouseup 58

## N

Nachfahrenmethoden 352  
Namensraum-Konflikte 330  
Namensraum-Verunreinigung 323  
Narrow Column 52  
Negation 28  
nextPage 250, 251  
Nicht-Klassenattribute 110  
notDeepEqual() 345  
notEqual() 345

## O

Objektinstanz 15  
Objektliterale 82, 141  
Objektorientierte Programmierung 329  
Objekt-Untersuchung 365  
ok() 344  
outerFn() 324, 325  
outerWidth() 97

## P

pageNum 243  
parseFloat() 284  
Pascal 324  
PHP 151  
Plug-in-Repository 175  
Plug-ins  
    Anforderungen 216  
    Architektur 175  
    Benutzerdefinierte Selektoren 180  
    Cycle 176  
    Einführung 9  
    Globale Funktionen 181  
    Suchen 175  
    Verwendung 176  
POST 155

prepRows() 293  
Promise-Objektmethoden 364  
Punktschreibweise 285  
<p> 114

## Q

Quicksort 282  
QUnit  
    Andere Testfunktionen 345  
    Ansynchrones JavaScript 344  
    Download 339  
    Einführung 216  
    Herunterladen 339  
    HTML-Testdokument 340  
    Praktische Überlegungen 346  
    Tests 341

## R

Refactoring 55  
Referenzzähler 334  
rel 110  
relatedTarget 246  
requestAnimationFrame() 265  
resize 58  
Rückgabewert 325

## S

Safari 24  
Safari Web Inspector 18, 24  
Sammlungsbearbeitung 353  
Schleifenprüfung 230  
Schlüssel-Wert-Paare 140  
ScriptDoc 217  
<script> 145, 318, 340  
scroll 58  
scroll-Eereignis 251  
scrollToVisible 252  
search 317  
Seiten laden 45  
select 58  
selected 296  
Selektorausdrücke  
    Andere benutzerdefinierte  
        Selektoren 351  
Attribute 350

- Selektorausdrücke (*Fortsetzung*)  
Einfaches CSS 349  
Formulare 351  
Position 350  
Position unter Geschwistern 350  
Position unter passenden Elementen 350
- Selektoren  
Anpassen 226  
Benutzerdefiniert 351  
Durchlaufen des DOM 219  
Eigenes Selektor-Plug-in 226  
Geschwindigkeit 230, 231  
Optimieren 226  
Performance 228  
Referenz 226
- Selektor-Performance  
Einführung 228  
Geschwindigkeit 230  
Sizzle 229
- Selenium 346
- Server 150
- SET 296, 298  
set 227  
setInterval() 255, 265  
Set-Methoden 128  
setTimeout 256  
setTimeout() 254  
setup() 343  
showBio() 270  
showDetails() 267  
Sizzle 219, 229
- Sjax 139
- Skripte 145  
sort 279  
sort-alpha 280  
Sortieralgorithmen 282  
sortKey 290  
sortKeys[keyType](\$cell) 286  
Speicherlecks 334, 336  
sp\$.event.special 255  
src 319  
start() 344  
stripe() 237  
submit 58  
Synchrone Aufrufe 139
- T
- Tabellenzeilen  
Sortieren 275, 276, 278  
Streifenmuster 223
- Tags 147  
Tastaturereignisse 76  
<tbody> 276  
teardown 256, 343  
Test-Driven Development 342  
Tests 341  
test() 341  
text-shadow 297  
this 53, 201, 343  
throttledScroll 256, 312  
<th> 289  
Timestamp 284  
title 110, 112  
toggleSwitcher 74  
Transport 308  
Traversierungsmethoden 226  
<tr> 292
- U
- Überoptimierung 228  
Umgekehrte Einfügemethoden 121  
Unendliches Scrollen 251  
Unit-Tests 216, 341  
unload 58
- V
- Verhaltenswarteschlangen 55  
Verkettung 10, 40  
Verhalten 202  
Verweisen auf Funktionen 47  
Verweisschleifen 335  
Verzögerte Objekte 268  
Animations-Promise 270  
Erstellen 364  
Methoden 364  
Methoden für Promise-Objekte 364  
Methoden verzögerter Objekte 364  
Objekterstellung 364  
Vordefinierte Effekte 357  
Vorfahrenmethoden 353

## W

Warteschlangen 261  
Manipulationsmethoden 358  
Web Inspector 24  
Wert-Callback 111  
Widgets  
    Aktivieren 213  
    Deaktivieren 213  
    Einführung 187, 209  
    Entfernen 212  
    Ereignis auslösen 215  
    Erstellen 210, 211  
    Optionen 213  
    Schaltfläche 187  
window.onload  
    Codeausführung 45, 46  
World Wide Web 7

## X

XML 146, 150  
XMLHttpRequest 135, 158, 276, 308

## Y

Yahoo! Query Language (YQL) 303  
YAML 313

## Z

Zeiger 334  
Zeilen erstellen  
    Einführung 290  
    Inhalt 293, 294  
    JSON-Objekt 292, 293  
Zusammengesetzte Effekte 92  
Zusammengesetzte Ereignisse  
    Anklickbare Elemente  
        hervorheben 60  
    Einführung 58  
    Erweiterte Funktionen 58  
    Handler 58

## Sonderzeichen

#id 349  
\$-Alias 193  
\$cells 235  
\$columnCells 235  
\$(document).ready() 45, 46, 47, 48  
    Argumente 329  
\$() 25, 114, 249, 295, 296  
\$.ajaxPrefilter() 318  
\$.ajaxPrefilter([dataTypes], handler) 363  
\$.ajaxSetup() 169  
\$.ajaxTransport(transportFunction) 363  
\$.ajaxTransport() 319  
\$.ajax(options) 362  
\$.ajax() 194, 306  
    Globale Funktion 168  
    Optionen 306  
\$.attrFn 296  
\$.attrHooks 296  
\$.contains(a,b) 354  
\$.cssHooks 296, 297  
\$.Deferred() 269  
\$.Deferred([setupFunction]) 364  
\$.each(collection, callback) 365  
\$.each() 194, 195, 291  
\$.extend(target, addition,...) 365  
\$.extend() 197, 227  
\$.getJSON(url, [data], [callback]) 362  
\$.getJSON() 141, 142, 293  
\$.getScript(url, [callback]) 362  
\$.getScript() 145, 166  
\$.get(url, [data], [callback],  
    [returnType]) 362  
\$.get() 147, 162  
\$.globalEval(code) 363  
\$.grep(array, callback, [invert]) 365  
\$.grep() 194  
\$.inArray(value, array) 365  
\$.isArray(object) 365  
\$.isEmptyObject(object) 365  
\$.isFunction(object) 365  
\$.isPlainObject(object) 365

\$.isWindow(object) 365  
\$.isXMLDoc(object) 365  
\$.makeArray(object) 365  
\$.map(array, callback) 365  
\$.map() 194  
\$.merge(array1, array2) 365  
\$.noConflict() 193  
\$.noConflict([removeAll]) 366  
\$.noop() 366  
\$.now() 366  
\$.param(map) 363  
\$.parseJSON(json) 363  
\$.parseXML(xml) 363  
\$.post(url, [data], [callback],  
    [returnType]) 362  
\$.propHooks 296  
\$.proxy(fn, context) 357  
\$.trim(string) 366  
\$.trim() 281  
\$.type(object) 365  
\$.unique(array) 365  
\$.valHooks 296  
\$.when(deferreds) 364  
\$.widget() 209  
\* 349  
.abort() 307  
.addClass(class) 359  
.addClass() 15, 109, 131, 295  
.after(content) 361  
.after() 132  
.ajaxComplete(handler) 363  
.ajaxError(handler) 363  
.ajaxSend(handler) 363  
.ajaxStart(handler) 363  
.ajaxStart() 160  
.ajaxStop(handler) 363  
.ajaxStop() 160  
.ajaxSuccess(handler) 363  
.always() 269  
.always([callback]) 364  
.andSelf() 234, 353  
.animate(attributes, options) 358  
.animate(attributes, [speed], [easing],  
    [callback]) 358  
.animate() 93, 99, 183, 261, 270, 295  
.appendTo(selector) 361  
.appendTo() 117, 132  
.append(content) 361  
.append() 132  
.attr(key) 359  
.attr(key, fn) 359  
.attr(key, value) 359  
.attr(map) 359  
.attr() 109, 296  
.before(content) 361  
.before() 132  
.bind(type, [data], handler) 354  
.bind() 189  
.blur(handler) 355  
.blur() 356  
.button() 187  
.change(handler) 355  
.change() 356  
.children([selector]) 352  
.class 349  
.clearQueue([queueName]) 358  
.click(handler) 355  
.click() 104, 243, 356  
.clone() 125  
.clone([withHandlers]) 362  
.closest(selector) 353  
.closest() 236, 245  
.column() 235  
.contents() 352  
.context 233  
.css(key) 360  
.css(key, value) 360  
.css(map) 360  
.css() 96, 295  
.cycle() 177  
.data(key) 362  
.data(key, value) 362  
.data() 289  
.dbclick(handler) 355  
.dblclick() 357  
.delay(duration, [queueName]) 358  
.delegate(selector, handlers) 355  
.delegate(selector, type, [data],  
    handler) 355  
.delegate() 71, 246

.dequeue 358  
.detach 132  
.detach([selector]) 361  
.die(type, [handler]) 354  
.die() 71  
.done(callback) 364  
.done() 269, 310, 320  
.each(callback) 354  
.each() 119, 235, 295  
.effect() 185  
.empty() 132, 361  
.end() 129, 353  
.eq(index) 352  
.eq() 231  
.error(handler) 355  
.error() 357  
.eval() 315  
.fadeIn('slow') 90  
.fadeIn([speed], [callback]) 357  
.fadeOut() 267  
.fadeOut([speed],[callback]) 358  
.fadeToggle([speed],[callback]) 358  
.fadeTo(speed, opacity,[callback]) 358  
.fail(callback) 364  
.fail() 269, 320  
.filter(callback) 352  
.filter(selector) 352  
.find(selector) 352  
.find() 129  
.first() 352  
.focusin(handler) 355  
.focusout(handler) 355  
.focus(handler) 355  
.focus() 357  
.getElementById() 229  
.getElementsByClassName() 229  
.getElementsByTagName() 229  
.getErrors() 315  
.get(index) 354  
.get() 354  
.hasClass(class) 359  
.has(selector) 352  
.height(value) 360  
.height() 360  
.hide(speed, [callback]) 357  
.hide('speed') 89  
.hide() 86, 88, 357  
.hover(enter, leave) 356  
.hover() 61, 244  
.html(value) 359  
.html() 128, 132, 359  
.index(element) 354  
.index() 354  
.innerHeight() 360  
.innerWidth() 360  
.insertAfter(selector) 361  
.insertAfter() 116, 132  
.insertBefore(content) 361  
.insertBefore() 116, 132  
.isRejected() 364  
.isResolved() 364  
.is(selector) 354  
.join() 122, 305  
.keydown(handler) 355  
.keydown() 357  
.keypress(handler) 355  
.keypress() 357  
.keyup(handler) 355  
.keyup() 357  
.last() 352  
.length 354  
.live(type, handler) 354  
.live() 71, 164, 247  
.load(handler) 355  
.load(url, [data], [callback]) 362  
.load() 138, 155, 302  
.map(callback) 353  
.mousedown(handler) 355  
.mouseenter(handler) 355  
.mouseleave(handler) 355  
.mousemove(handler) 356  
.mouseout(handler) 356  
.mouseover(handler) 356  
.mouseup(handler) 356  
.nextAll() 233  
.nextAll([selector]) 352  
.nextUntil([selector],[filter]) 352  
.next([selector]) 352  
.not(selector) 352  
.offsetParent() 353  
.offset() 252, 360  
.one(type, [data], handler) 354

.outerHeight(includeMargin) 360  
.outerWidth(includeMargin) 360  
.parentsUntil([selector], [filter]) 353  
.parents([selector]) 353  
.parent([selector]) 353  
.pipe([doneFilter], [failFilter]) 364  
.position() 360  
.prependTo(selector) 361  
.prependTo() 117, 132  
.prepend(content) 361  
.prepend() 132  
.prevAll([selector]) 353  
.prevObject 233  
.prevUntil([selector],[filter]) 353  
.prev([selector]) 353  
.promise() 269  
.promise([queueName], [target]) 358  
.promise([target]) 364  
.prop(key) 359  
.prop(key, fn) 359  
.prop(key, value) 359  
.prop(map) 359  
.prop() 114, 295  
.pushStack(elements) 353  
.pushStack() 236  
.push() 281  
.querySelectorAll() 229  
.queue 358  
.queue() 105  
.ready(handler) 354  
.ready() 16, 48, 49  
.rejectWith(context, [args]) 364  
.reject([args]) 364  
.removeAttr(key) 359  
.removeAttr() 109  
.removeClass(class) 359  
.removeClass() 15, 109  
.removeData(key) 362  
.removeProp(key) 359  
.remove() 132  
.remove([selector]) 361  
.replaceAll(selector) 361  
.replaceAll() 132  
.replaceWith(content) 361  
.replaceWith() 132  
.resize(handler) 356  
.resolveWith(context, [args]) 364  
.resolve([args]) 364  
.scrollLeft(value) 360  
.scrollLeft() 360  
.scrollTop(value) 360  
.scrollTop() 252, 360  
.scroll(handler) 356  
.selector 233  
.select(handler) 356  
.select() 357  
.send() 319  
.serializeArray() 363  
.serialize() 157, 363  
.show(speed, [callback]) 357  
.show('fast') 89  
.show('normal') 89  
.show('slow') 89  
.show('speed') 89  
.show() 86, 88, 357  
.siblings([selector]) 353  
.slice(start, [end]) 352  
.slideDown() 91, 263  
.slideDown([speed],[callback]) 357  
.slider() 188  
.slideToggle() 92  
.slideToggle([speed],[callback]) 357  
.slideUp() 91  
.slideUp([speed],[callback]) 357  
.sort() 280  
.splice() 281  
.status 307  
.stopPropagation() 66  
.stop() 263  
.stop([clearQueue], [jumpToEnd]) 358  
.submit(handler) 356  
.submit() 357  
.switchClass() 200  
.tar.gz-Datei 339  
.text(value) 359  
.text() 113, 129, 132, 296, 359  
.then(doneCallbacks, failCallbacks) 364  
.toArray() 354  
.toggleClass(class) 359  
.toggleClass() 60, 109  
.toggle(handler1, handler2, ...) 356  
.toggle() 59

.toggle([speed], [callback]) 357  
.triggerHandler(type, [data]) 356  
.trigger(type, [data]) 356  
.trigger() 75, 253  
.unbind() 74  
.unbind([type], [handler]) 354  
.undelegate(selector, type, [handler]) 355  
.undelegate() 71  
.unload(handler) 356  
.unwrap() 361  
.val(value) 359  
.val() 295, 359  
.width(value) 360  
.width() 360  
.wrapAll(content) 361  
.wrapAll() 119, 132  
.wrapInner(content) 361  
.wrapInner() 131, 132, 278  
.wrap(content) 361  
.wrap() 119, 132  
.zip-Datei 339  
.animated 351  
.button 36, 351  
.checkbox 351  
.checked 36, 351  
.contains(text) 351  
.disabled 36, 351  
.empty 351  
.enabled 36, 351  
.eq(index) 350  
.even 350  
.file 351  
.first 350  
.first-child 350  
.focus 351  
.gt(index) 350  
.has(a) 351  
.header 351  
.hidden 351  
.image 351  
.input 36, 351  
.last 350  
.last-child 350  
.lt(index) 350  
.not(a) 350  
.nth-child(even) 350  
.nth-child(formula) 350  
.nth-child(index) 350  
.nth-child(odd) 350  
.odd 350  
.only-child 350  
.parent 351  
.password 351  
.radio 351  
.reset 351  
.selected 36, 351  
.submit 351  
.text 351  
.visible 351  
[attr] 350