open source library

Dave Crane
Bear Bibeault
Jord Sonneveld

Ajax in practice

Das Praxisbuch für die Web 2.0-Entwicklung

ADDISON-WESLEY

Teil II

Bewährte Ajax-Techniken

Teil II umfasst neun Kapitel, die jeweils einen der für Ajax-Programme wichtigen Bereiche der Entwicklung von Webanwendungen behandeln.

Kapitel 5 eröffnet Teil II mit einem tiefgehenden Blick auf die Ereignisbehandlung in Browsern. Wir besprechen die Modelle der Ereignisbehandlung und erklären, wie Sie die Ereignishandler für die verschiedenen Ereignistypen einrichten. Anschließend zeigen wir Browserprobleme auf und geben Möglichkeiten an, um die browserübergreifende Codierung zu vereinfachen.

In *Kapitel 6* entwickeln wir Möglichkeiten, um die Eingabe von Formulardaten mit Hilfe der im vorhergehenden Kapitel besprochenen Ereignisbehandlung zu validieren. Wir untersuchen ein Validierungsframework, und Sie

erfahren, wie Sie Formularübertragungen abfangen, um eine vollständige Seitenaktualisierung zu vermeiden.

Navigation in den Inhalten der Anwendung ist das Thema von *Kapitel 7*. Menüs, Bäume, Akkordeon-Steuerelemente, Registerkarten und Symbolleisten werden besprochen. Im Beispielcode zu diesem Kapitel schnüffeln wir auch kurz in OpenRico und qooxdoo hinein.

Der Umgang mit Benutzern, die darauf bestehen, im Browser auf die Steuerelemente ZURÜCK oder AKTUALISIEREN zu klicken, wird in *Kapitel 8* angegangen. Wir beschreiben Verfahrensweisen, um diese Steuerelemente zu verbergen, zeigen aber auch, wie Sie mit ihnen umgehen, wenn sie nicht verborgen sind.

Kapitel 9 konzentriert sich auf das Einrichten von Drag-&-Drop-Fähigkeiten zu Webanwendungen. Wir lassen uns von der Bibliothek Scriptaculous helfen, um Listen mit Hilfe von Drag & Drop zu sortieren, und untersuchen einen einfachen Warenkorb mit Drag & Drop. Ein Blick auf ICEfaces rundet dieses Kapitel ab.

Usability-Aspekte, vor allem die gewöhnlich mit Ajax-Anwendungen verbundenen, werden in *Kapitel 10* besprochen. Wir entwickeln Strategien, um Latenzprobleme anzupacken und erklären, wie mögliche frustrierende Erfahrungen für den Benutzer vermieden werden können.

In *Kapitel 11* geht es um die Beibehaltung des Clientstatus, um Caching und den Vorabruf von Daten sowie andere Themen aus dem Gebiet der Statusverwaltung.

Offene APIs für spannende Webservices wie Yahoo! Maps, Yahoo! Geocoding, Yahoo! Traffic, die Google-Suche und Flickr-Fotodienste sind Gegenstand von *Kapitel 12*. Wir entwickeln Maßnahmen, um die gefürchteten browserübergreifenden Sicherheitsgrenzen zu umgehen, und Sie erfahren, wie Sie mit Ajax REST-Anfragen an diese Dienste senden.

Abschließend werden in *Kapitel 13* alle Themen zusammengefasst, um eine funktionierende Webanwendung zu erstellen, die die offenen APIs von Yahoo! und Flickr verwendet.

5 Ereignisse handhaben

Dieses Kapitel behandelt folgende Themen:

- Modelle zur Ereignisbehandlung durch Browser
- Häufig vorkommende Ereignistypen
- Die Vereinfachung der Ereignisbehandlung
- Ereignisbehandlung in der Praxis

Seit Ajax uns ermöglicht, Webanwendungen mit einem hohen Maß an Interaktivität zu erstellen, die flüssig auf Benutzeraktionen reagieren, ist die Zeit langweiliger HTML-Anwendungen vorbei. Bei diesen Aktionen kann es sich darum handeln, auf eine Schaltfläche zu klicken, etwas in ein Textfeld einzugeben oder einfach die Maus zu bewegen. Seit es grafische Benutzerschnittstellen (Graphical User Interfaces, GUIs) gibt, werden Benutzeraktionen in Ereignisse umgesetzt, und in der Welt der Browser ist es nicht anders. Unternimmt ein Benutzer auf einer Webseite etwas, werden in der DOM-Hierarchie, mit der er interagiert, Ereignisse ausgelöst, und wenn mit den Ereignissen für die Elemente des Dokuments Ereignishandler verknüpft sind, werden diese beim Eintritt eines Ereignisses aufgerufen. Ajax-Anwendungen stützen sich in hohem Maß auf Ereignisse und Ereignishandler; sie können sogar als Lebensader aller Ajax-Anwendungen gelten.

Sehen wir uns an, wie wir einen einfachen Ereignishandler in eine Webseite einbauen können, bevor wir zu weit vorgreifen. Achten Sie im folgenden Code auf das Attribut `onclick` des Elements `<img>`. Es definiert einen Ereignishandler, der vom Browser aufgerufen wird, wenn der Benutzer auf das Element klickt.

```
<html>
  <body>
    <img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
  </body>
</html>
```

Wenn Sie dieses Beispiel in einen Browser laden, sehen Sie, dass die Warnmeldung »Woof!« (Abb. 5.1) eingeblendet wird, sobald Sie klicken, während Sie mit der Maus über das Bild fahren.

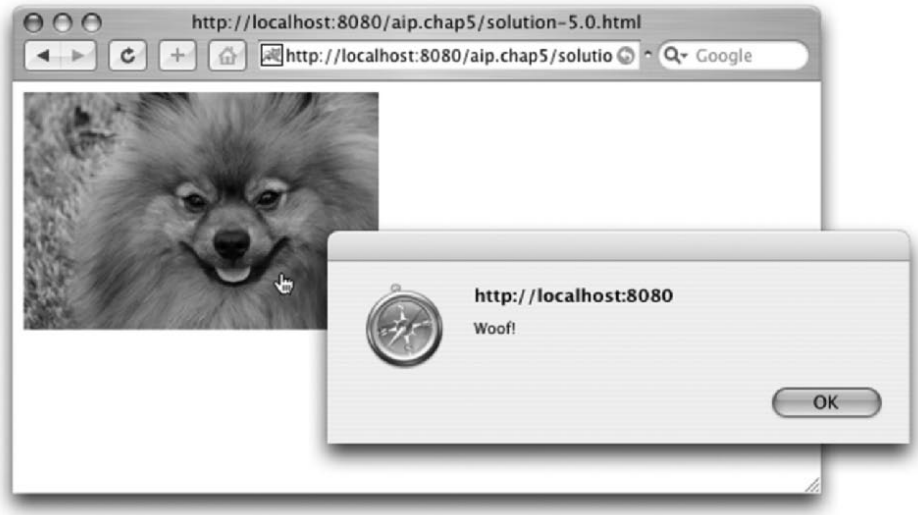

Abbildung 5.1: Wie der Hund zum Bellen gebracht wird

Daran erkennen Sie, wie einfach es ist, einem DOM-Element einen Ereignishandler zuzuweisen. Im Verlauf dieses Kapitels untersuchen wir die wichtigsten Aspekte der Ereignisbehandlung. Wir beginnen mit den unterschiedlichen Methoden zur Definition von Ereignishandlern anhand der zur Verfügung stehenden unterschiedlichen Modelle. Dann schauen wir uns an, wie sich der Vorgang bei den verschiedenen Browserplattformen unterscheidet, und befassen uns damit, wie er sich auf Browserimplementierungen portieren lässt. Außerdem erfahren wir, wie dem Ereignishandler die Informationen über das Ereignis übermittelt werden, wenn er aufgerufen wird. Wir erörtern die Konzepte *Ereignisaufstieg* und *Ereignisaufzeichnung*, die bestimmen, wie Ereignisse durch das DOM weitergeleitet werden, und sehen uns die häufigsten Ereignistypen genauer an. Schließlich bringen wir einige Beispiele aus der Praxis, die zeigen, wie wir diese Konzepte in unseren Anwendungen umsetzen können.

5.1 Modelle zur Ereignisbehandlung

Nachdem wir gesehen haben, wie leicht sich einfache Ereignishandler *deklarieren* lassen, nehmen Sie sicher an, dass es ebenso einfach sein sollte, sie zu schreiben. Wir fügen einfach ein Skript in die Handlerattribute ein, das der Browser dann ausführt, wenn das Ereignis eintritt. Was kann einfacher sein? Wäre es wirklich so, brauchten wir aber dieses Kapitel nicht, oder?

Heute müssen wir uns mit drei Ereignismodellen auseinandersetzen, um in unseren Webanwendungen Ereignisse verwenden zu können:

- Das grundlegende Ereignismodell, das informell auch als DOM-Level-0-Modell bezeichnet wird, das ziemlich einfach, unkompliziert und einigermaßen plattformunabhängig ist.
- Das DOM-Level-2-Ereignismodell, das mehr Flexibilität bietet, aber nur von standardkonformen Browsern wie Firefox, Mozilla und Safari unterstützt wird.
- Das Internet Explorer-Ereignismodell, das in der Funktion DOM Level 2 ähnelt, aber ein proprietäres Modell von Internet Explorer ist.

Zuerst sehen wir uns an, wie man mit dem Grundmodell Handler registriert und schreibt, danach kümmern wir uns um die beiden anderen Modelle.

5.1.1 Grundlegende Ereignisbehandlung

Das Beispiel, das wir in der Einleitung betrachtet haben, veranschaulicht die Verwendung des Grund- oder DOM-Level-0-Modells, das den ältesten Ansatz der Ereignisbehandlung darstellt und sich weitgehender, wenn auch nicht vollständiger Plattformunabhängigkeit erfreut. Es eignet sich gut für Ereignisbehandlung mit einfachen Ansprüchen und wird, wie wir sehen werden, von den fortgeschritteneren Modellen nicht vollkommen ersetzt, sondern üblicherweise in Verbindung mit ihnen benutzt.

Dieses Modell ermöglicht zwei Vorgehensweisen beim Zuweisen von Ereignishandlern:

- Innerhalb der Zeile (inline) mit Hilfe des HTML-Element-Markups durch die Ereignisattribute der HTML-Elemente
- Skriptgesteuert durch die Eigenschaften der DOM-Elemente

Denken Sie an das Element `<img>` aus unserem Beispiel zurück:

```
<img src="image.jpg" id="anImage" onclick="alert('Woof!');"/>
```

Es verwendet die *Inline*-Technik.

Der Wert des Ereignisattributs `onclick` wird zum Rumpf einer anonymen Funktion, die als Handler für das Klickereignis dient. Das ist zwar einfach, hat aber Grenzen.

Der bewährte Designansatz für Webanwendungen trennt das Aussehen der Anwendung (HTML) von ihrem Verhalten (JavaScript). Der Inline-Ansatz für die Definition von Ereignishandlern verletzt dieses Prinzip, weshalb im Allgemeinen empfohlen wird, die Verwendung solcher Handlerdeklarationen zu beschränken oder zu vermeiden.

Besser ist es, den Ereignishandler skriptgesteuert an das DOM-Element zu binden. Diese Technik ist in den letzten Jahren wegen der stärkeren Standardisierung des DOM für Browser und der zunehmenden Vertrautheit der JavaScript-Entwickler damit auf dem Vormarsch. Alle DOM-Elemente weisen Eigenschaften für die Ereignisse auf, die für das Element ausgelöst werden können: beispielsweise `onclick`, `onkeyup` oder `onchange`.

Arbeiten wir den gezeigten Beispielcode zu einem vollständigen HTML-Dokument um, in dem wir den Ereignishandler `onclick` des Bildes per Programm hinzufügen, wie es in Listing 5.1 geschieht:

Listing 5.1: Zuweisen eines Ereignishandlers durch ein Skript

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {      ❶ Deklariert den Ereignishandler onload der Seite
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>     ❷ Deklariert das skriptfreie Bildelement
  </body>
</html>
```

Wenn Sie den Quellcode zu diesem Kapitel von der Adresse *www.manning.com/crane2* heruntergeladen haben, finden Sie dieses HTML-Dokument in der Datei `chap5/listing-5.1.html`.

Dieses Beispiel ist zwar von der Funktion her mit dem vorherigen gleichwertig, zeigt jedoch mehr Raffinesse. Wir haben das Verhalten vom Aussehen getrennt, indem wir das Skript aus dem Element `<body>` ❷ in ein Element `<script>` im Handler `<head>` ❶ für die Seite verlagert haben. Beachten Sie, dass wir den Code in einer weiteren Ereignishandlerfunktion untergebracht haben: nämlich im Ereignishandler `onload` ❶ für die Seite.

Obwohl diese Technik nach mehr Code für dieselbe Sache aussieht, verbessert sie nicht nur die Struktur der Seite, sondern gibt uns auch mehr Flexibilität. Ein wichtiger Aspekt dieser Flexibilität ist die Möglichkeit zu steuern, *wann* Handler eingerichtet und entfernt werden. Bei der Inline-Methode sind wir darauf beschränkt, Handler beim Laden der Seite einzurichten, die dann bestehen, solange es die Seite gibt. Die skriptgesteuerte Zuweisung eines Handlers ermöglicht uns, einen Handler zu erstellen, wann immer wir wollen. In Listing 5.1 haben wir uns entschieden, es beim Laden der Seite zu tun, aber wir hätten es genauso gut auf einen späteren Zeitpunkt verschieben können, als Ergebnis eines anderen Ereignisses. Außerdem können wir den Ereignishandler jederzeit *löschen*, indem wir der Ereigniseigenschaft den Wert `null` zuweisen – was bei Inline-Handlern nicht möglich ist.

In unserem Beispiel haben wir den Handler mit einem anonymen Funktionsliteral erstellt – warum sollten wir auch eine eigene benannte Funktion schreiben, wenn es nicht erforderlich ist? Bei der Zuweisung benannter Funktionen als Ereignishandler

müssen Sie jedoch unbedingt darauf achten, keine Klammern hinter den Funktionsnamen zu setzen. Wir wollen ja einen Verweis auf die Funktion als Eigenschaftswert zuweisen, nicht das Ergebnis der *Aufrufs* der Funktion! Die folgende Zeile ruft zum Beispiel eine benannte Funktion `sayWoof()` auf, anstatt sie als Ereignishandler zu deklarieren. Vermeiden Sie diesen häufigen Fehler!

```
Element.onclick = sayWoof(); //Falsch!

Element.onclick = sayWoof;   //Richtig!
```

Obwohl das DOM-Level-0-Ereignismodell ein gewisses Maß an Flexibilität aufweist, leidet es unter Einschränkungen; es erlaubt beispielsweise keine einfache Verkettung von JavaScript-Funktionen als Reaktion auf ein Ereignis.

Wie sehen wir also zwei Funktionen zur Behandlung eines Ereignisses vor? Nutzen wir zunächst einen ziemlich naiven Ansatz und ändern unser Beispiel, indem wir der Eigenschaft `onclick` des Elements `<img>` zwei Ereignishandler hinzufügen wie in Listing 5.2 (in der Datei `chap5/listing-5.2.html` des Quellcodes zum Herunterladen). Die hinzugefügten Stellen sind durch Fettschrift hervorgehoben.

Listing 5.2: Versuch der Zuweisung von zwei Handlern

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick = function() {
          alert('Woof!');
        }
        document.getElementById('anImage').onclick = function() {
          alert('Woof again!');
        }
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

Führen wir diesen Code aus, wird offensichtlich nur der zweite Handler aufgerufen, weil nur eine einzige Warnmeldung mit dem Inhalt »Woof again!« erscheint. Dies sollte bei näherer Betrachtung des Codes nicht überraschen. Da `onclick` einfach eine Eigenschaft des Elements `<img>` ist, überschreibt jede weitere Zuweisung wie bei allen Eigenschaften die vorhergehende.

Damit stellt sich eine interessante Frage: Ist es möglich, mehrere Funktionen als Reaktion auf ein Ereignis zuzuweisen? Nach dem DOM-Level-0-Modell gibt es kein Verfahren, durch Zuweisen von Handlern zu den Ereigniseigenschaften des Elements mehrere Handler für dasselbe Ereignis zu verketten. Wir könnten den Code mehrerer Funktionen zu einer zusammenfassen oder eine Funktion schreiben, die die anderen nacheinander aufruft. Diese Ansätze sind jedoch alle ziemlich umständlich und nicht gut skalierbar. Bietet sich kein anderer Ausweg, wäre das Muster des Beobachters eine Möglichkeit (das auch unter dem Namen Verleger/Abonnent bekannt ist), bei dem der registrierte Handler als Beobachter dient und die übrigen Funktionen sich als Abonnenten registrieren können.

Glücklicherweise brauchen wir nicht zu derartigen Tricks greifen, da die Browser die Registrierung mehrerer Handler zulassen, wenn wir die fortgeschrittenen Modelle der Ereignisbehandlung verwenden – leider aber nicht browserunabhängig. Sehen wir uns an, wie das geht.

5.1.2 Erweiterte Ereignisbehandlung

In einer idealen Welt würde Code, der für einen Browser geschrieben wurde, problemlos auch in allen anderen funktionieren. Unsere Welt ist aber nicht ideal, so dass wir uns mit Unterschieden zwischen Browsern befassen müssen, wenn wir zu den fortgeschrittenen Ereignismodellen kommen. Auf der einen Seite gibt es die Methode des W3C (World Wide Web Consortium), auf der anderen Seite die von Microsoft. Sehen wir uns zunächst die standardisierte W3C-Methode an.

Für Browser nach dem DOM-Level-2-Ereignismodell ist für jedes DOM-Element eine Methode `addEventListener()` definiert, die aufgerufen werden kann, um dem Element einen Ereignishandler hinzuzufügen. Sie übernimmt drei Argumente: einen String, der den Ereignistyp deklariert, die auszuführende Handlerfunktion (auch als *Listener* bezeichnet) und einen Booleschen Wert, der besagt, ob die Ereignisaufzeichnung aktiviert werden soll. Das letzte Argument werden wir im Zusammenhang mit der Ereignisweiterleitung erläutern und belassen es für den Augenblick einfach auf der Einstellung `false`.

Das Argument für den Ereignistyp erwartet einen String, der den Namen des zu beobachtenden Ereignistyps enthält. Dabei handelt es sich um den Attributnamen für das Ereignis ohne das Präfix `on` – zum Beispiel `click` oder `mouseover`.

Ändern wir nun den Beispielcode aus Listing 5.2 so, dass er diese Methode verwendet. Wir ersetzen das einfache Verfahren (das die Eigenschaft `onclick` des Elements festlegt) durch Aufrufe der Methode `addEventListener()` wie in Listing 5.3 (Änderungen durch Fettschrift hervorgehoben):

Listing 5.3: Einen Ereignishandler nach der W3C-Methode hinzufügen

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof!'); },
          false);
        document.getElementById('anImage').addEventListener(
          'click',
          function() { alert('Woof again!'); },
          false);
      };
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

Wird *diese* Seite angezeigt und klicken Sie auf das Bild, erscheinen beide Meldungsfelder, ohne dass Sie sich merkwürdiger Containerfunktionen bedient haben, um die beiden Ereignishandler zu verketten. Beachten Sie, dass dieser Code im Internet Explorer nicht funktioniert; weiter hinten in diesem Abschnitt erfahren Sie, wie IE die erweiterte Ereignisbehandlung auf seine proprietäre Art implementiert.

Beachten Sie außerdem, dass das DOM-Level-2-Ereignismodell nicht garantiert, in welcher Reihenfolge die Handler ausgeführt werden, wenn Sie wie in unserem Beispiel mehrere Handler für dasselbe Ereignis und dieselben Elemente erstellt haben. Beim Testen gab es Beobachtungen, dass sie anscheinend in der Reihenfolge der Einrichtung aufgerufen werden, aber dafür gibt es keine Garantie, und es wäre unklug, sich darauf zu verlassen.

Um einen Ereignishandler aus einem Element zu löschen, steht die für DOM-Elemente definierte Methode `removeEventListener()`zur Verfügung.

Das proprietäre Mittel von Microsoft zum Einrichten von Ereignishandlern ist konzeptionell ähnlich, unterscheidet sich aber in der Implementierung. Es nutzt eine für DOM-Elemente definierte Methode `attachEvent()`. Diese Funktion übernimmt zwei Argumente: den Ereignisnamen und die auszuführende Ereignishandlerfunktion. Anders als der Ereignistyp, der bei `addEventListener()` benutzt wird, erwartet sie den vollständigen Namen der Ereigniseigenschaft, also mit dem Präfix `on`.

Ausgestattet mit diesem Wissen wollen wir unseren Beispielcode noch einmal ändern. Wir fügen eine Erkennung ein und verwenden die für den benutzten Browser

passende Methode. Den aktualisierten Code finden Sie in Listing 5.4 (verfügbar im herunterladbaren Quellcode zu diesem Kapitel), die Änderungen sind wiederum durch Fettschrift hervorgehoben:

Listing 5.4: Auf beide Arten

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        if (document.getElementById('anImage').attachEvent) {
          document.getElementById('anImage').attachEvent(
            'onclick',
            function() { alert('Woof!'); });
          document.getElementById('anImage').attachEvent(
            'onclick',
            function() { alert('Woof again!'); });
        }
        else {
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof!'); },
            false);
          document.getElementById('anImage').addEventListener(
            'click',
            function() { alert('Woof again!'); },
            false);
        }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
  </body>
</html>
```

In der ersten Zeile des Ereignishandlers `onload` prüfen wir, welche Methode verwendet werden soll. Dieser Test wird als *Objekterkennung* bezeichnet. Anstatt auf einen bestimmten Browser zu testen, suchen wir nach der proprietären Methode `attachEvent()` für das Element. Ist sie vorhanden, benutzen wir sie, andernfalls die standardisierte W3C-Methode.

Wenn wir diese Seite in einem beliebigen Browser darstellen, ist garantiert, dass sie funktioniert, solange der Browser einen der beiden Mechanismen unterstützt. Klicken wir im Internet Explorer auf das angezeigte Bild, machen wir eine merkwürdige Beobachtung: Die Meldungsfelder erscheinen in umgekehrter Reihenfolge! Um bei der Wahrheit zu bleiben, wissen wir wie beim DOM-Level-2-Ereignismodell nicht, welche

Reihenfolge gilt. Die Definition der Methode `attachEvent()` besagt deutlich, dass mehrere mit demselben Ereignistyp für ein Element verknüpfte Handler in zufälliger Reihenfolge ausgelöst werden.

Damit ist unsere Erforschung der Methoden abgeschlossen, mit denen Ereignishandler in verschiedenen Browsern verkettet werden können. Sie haben gesehen, wie leicht die Inline-Technik zu benutzen ist, aber auch deren Nachteile erfahren. Die DOM-Level-0-Methode der Registrierung von Ereignishandlern lässt sich auf andere Browser portieren, bietet aber keine Automatik für die Verkettung mehrerer Ereignishandlerfunktionen. Wir haben Ihnen gezeigt, wie Sie Ereignishandler mit dem DOM-Level-2- oder dem Internet Explorer-Modell auf fortgeschrittenere Art einrichten können. Dieser Ansatz ist zwar flexibel und erlaubt das dynamische Verknüpfen, Löschen und Verketten von Ereignishandlern, leidet jedoch unter Problemen aufgrund der Unterschiedlichkeit von Browsern, die uns zwingen, Zuflucht zur Objekterkennung zu nehmen, um die für den jeweiligen Browser geeignete Methode zu wählen. Glücklicherweise gibt es Frameworks, die all diese Unterschiede ignorieren und uns helfen, Code zu schreiben, der sich auf alle unterstützten Browser portieren lässt. Wie uns Prototype auf diese Art Hilfestellung gibt, erfahren Sie in Abschnitt 5.3.

Lassen Sie uns vorher unsere allgemeinen Grundlagen der Ereignisbehandlung erweitern. In den nächsten Abschnitten lesen Sie Genaueres darüber, wie die Ereignisinformationen dem Handler übermittelt werden und wie Ereignisse innerhalb der DOM-Struktur weitergeleitet werden.

5.2 Das Event-Objekt und die Ereignisweiterleitung

Zwei weitere wichtige Themen für den Umgang mit Ereignissen im Browser sind das `Event`-Objekt und die Art der Weiterleitung von Ereignissen. Das `Event`-Objekt, eigentlich eine Instanz der Klasse `Event`, spielt beim Übertragen von Informationen über das Ereignis eine Rolle, und die Weiterleitung definiert, in welcher Reihenfolge ein Ereignis an seine Beobachter übertragen wird. Nehmen wir uns zuerst das `Event`-Objekt vor.

5.2.1 Das Event-Objekt

Wenn ein Ereignis ausgelöst wird, wird eine Instanz der Klasse `Event` angelegt, die durch eine Reihe interessanter Eigenschaften beschrieben wird. Mit unseren Ereignishandlern wollen wir normalerweise auf dieses `Event`-Objekt zugreifen, um Eigenschaften wie das HTML-Element abzufragen, bei dem das Ereignis aufgetreten ist, oder (bei Mausereignissen) welche Maustaste betätigt wurde. Die Objektinstanz wird den Ereignishandlern wie häufig im Zusammenhang mit Ereignissen auf browserspezifische Art zugänglich gemacht.

Bei standardkonformen Browsern wird die Instanz des `Event`-Objekts der Ereignishandlerfunktion als erster Parameter übergeben, im Internet Explorer wird sie als Eigenschaft an die Instanz des Fensterobjekts angehängt (im Grunde eine globale Variable).

Untersuchen wir, was wir zum Umgang mit diesem Objekt brauchen. Von den Meldungsfeldern haben wir genug; ändern wir also den Code so, dass Diagnoseinformationen in ein `<div>`-Element unterhalb des Bildes geschrieben werden, wie Sie es in Listing 5.5 sehen.

Listing 5.5: Abfragen der `Event`*-Instanz*

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {
        document.getElementById('anImage').onclick =
          function(event) {
            if (!event) event =     ❶ Fragt Instanz des Ereignisobjekts ab
              window.event;

            var target =     ❷ Ermittelt Verweis auf das Zielobjekt des Ereignisses
              event.target ? event.target : event.srcElement;
            document.getElementById('info').innerHTML +=
              'I woof at ' + target.id + '!<br/>';
          }
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

In diesem Beispiel erhalten wir einen Verweis auf die Instanz von `Event`, indem wir zunächst prüfen, ob der Parameter definiert ist, der der klugerweise `event` genannten Ereignishandlerfunktion übergeben wurde (was bei standardkonformen Browsern zutrifft); wenn dies nicht der Fall ist, kopieren wir die Eigenschaft `event` aus dem Fensterobjekt ❶, in dem IE sie abgelegt hat.

Anschließend möchten wir einen Verweis auf das *Zielobjekt* erlangen ❷ – also für das Element, für das das Ereignis erstellt wurde. Auch diesmal müssen wir browserspezifisch vorgehen, weil es zwischen der Klasse `Event` in Internet Explorer und in standardkonformen Browsern Unterschiede gibt.

Wir prüfen, ob die Standardeigenschaft `target` definiert ist, und benutzen andernfalls die proprietäre Eigenschaft `srcElement`.

Wie umständlich! Anscheinend müssen wir bei fast allen einzelnen Schritten der Ereignisbehandlung zweigleisig vorgehen, damit sie sowohl in Internet Explorer als auch in den Browsern funktioniert, die die W3C-Standards unterstützen.

Nun ja, das ist weitgehend so. Aber keine Angst; Hilfe naht bereits. Finden wir aber zunächst heraus, was es mit der Ereignisweiterleitung auf sich hat.

5.2.2 Ereignisweiterleitung

Bisher haben wir uns auf Handler konzentriert, die direkt für die Elemente definiert sind, die die Ereignisse auslösen, als wären dies die einzigen Handler von Bedeutung. Wie sich herausstellt, ist dies aber nicht so, sondern das Ereignis wird nicht nur an das Zielelement übertragen, sondern möglicherweise auch an alle seine Vorfahren in der DOM-Struktur. In diesem Abschnitt erfahren wir, wie Ereignisse innerhalb der DOM-Struktur weitergeleitet werden und wie wir beeinflussen können, welche Ereignishandler auf dem Weg aufgerufen werden – und sogar, wie wir die Weiterleitung eines Ereignisses steuern können.

Wir beginnen mit der Weiterleitung von Ereignissen in Browsern nach dem DOM-Level-2-Ereignismodell. Anschließend sehen wir, wie Internet Explorer nur einen Teil dieses Modells unterstützt.

Wenn in standardkonformen Browsern nach dem DOM-Level-2-Modell ein Ereignis ausgelöst wird, wird es in drei Phasen behandelt, die *Aufzeichnungs-*, *Ziel-* und *Aufstiegs*phase heißen.

In der Aufzeichnungsphase durchquert das Ereignis die DOM-Struktur vom Stammelement des Dokuments abwärts bis zum Zielelement. Alle für die durchlaufenen Elemente und den weiterzuleitenden Ereignistyp erstellten Ereignishandler werden aufgerufen, wenn der Handler als *Aufzeichnungshandler* registriert ist. Erinnern Sie sich an diesen dritten Parameter der Methode `addEventListener()`, den wir bisher ignoriert haben? Ist er auf `true` gesetzt, ist der Ereignishandler als *Aufzeichnungshandler* registriert, ist er auf `false` gesetzt, wie wir es bisher gehalten haben, als *Aufstiegshandler*. Ein Ereignishandler kann Aufzeichnungs- oder Aufstiegshandler sein, aber nicht beides.

Sobald das Ereignis das Zielelement erreicht und dabei alle passenden Aufzeichnungshandler auf dem Weg aktiviert hat, tritt die Weiterleitung in die Zielphase ein. In dieser Phase werden die für das Zielelement selbst angelegten Ereignishandler nach Bedarf ausgelöst. Gibt es sowohl einen Aufzeichnungs- als auch einen Aufstiegshandler, werden auch beide aufgerufen.

Anschließend kehrt die Weiterleitung die Richtung um und »blubbert« die DOM-Struktur aufwärts vom Ziel- zum Stammelement. Dies ist die Aufstiegsphase, in der alle für den Ereignistyp und die durchlaufenen Elemente vorhandenen Aufstiegshandler ausgelöst werden.

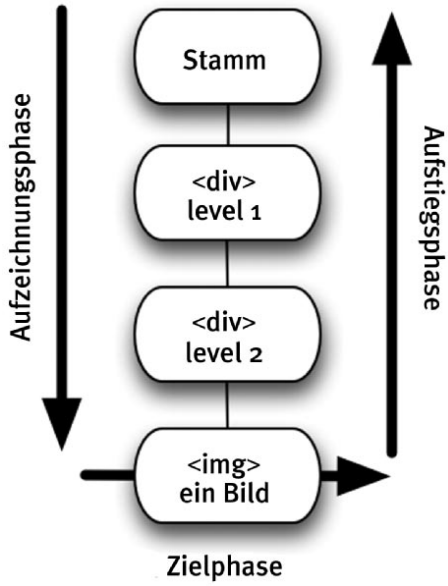

Abbildung 5.2: In der DOM-Struktur nach unten und wieder nach oben

Genug der Worte – wie wär's mit einem Bild? Ändern wir den Rumpf unseres Beispielprogramms so, dass das Element `<img>` von zwei `<div>`-Elementen umschlossen wird:

```
<div id="level1">
  <div id="level2">
    <img src="image.jpg" id="anImage"/>
  </div>
</div>
```

Wenn wir auf das Bildelement klicken, wird das Klickereignis, wie in Abb. 5.2 gezeigt, innerhalb der DOM-Struktur weitergeleitet.

Sehen wir uns das Ganze in Aktion an. Betrachten Sie den Code in Listing 5.6.

Listing 5.6: Erstellen von Aufzeichnungs- und Aufstiegshandlern

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript">
      window.onload = function() {    ❶ Legt Handler an
        document.getElementById('anImage').addEventListener(
          'click', react, false);
        document.getElementById('level1').addEventListener(
          'click', react, true);
```

```
        document.getElementById('level2').addEventListener(
          'click', react, false);
      }

      function react(event) {    ❷ Definiert Handlerfunktion
        document.getElementById('info').innerHTML +=
          'I woof at ' + event.currentTarget.id + '!<br/>';
      }
 </script>
  </head>
  <body>
    <div id="level1">    ❸ Definiert verschachteltes Element
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

Hier haben wir den Rumpf ❸ – wie vorhin beschrieben – geändert, indem wir das Element `<img>` zwischen zwei `<div>`-Elementen verschachtelt haben.

Innerhalb des Ereignishandlers `onload` ❶ erstellen wir drei Ereignishandler: einen für das Element `<img>` und je einen für die beiden umschließenden `<div>`-Elemente. Beachten Sie, dass der Ereignishandler für das Element mit der Bezeichnung `level1` durch den dritten Parameter als Aufzeichnungshandler registriert ist.

Allen Ereignishandlern wird dieselbe Funktion `react()` zugewiesen ❷, die eine Meldung mit dem Wert der Eigenschaft `currentTarget` der übergebenen Instanz von `Event` ausgibt. Diese Eigenschaft unterscheidet sich von der Eigenschaft `target` dadurch, dass `target` das Element bezeichnet, das das Ereignis ausgelöst hat, während `currentTarget` das derzeitige Subjekt der Ereignisweiterleitung angibt – anders ausgedrückt: das Element, für das der Ereignishandler erstellt wurde.

Raten Sie, in welcher Reihenfolge die Handler aufgerufen werden, bevor Sie einen Blick auf Abb. 5.3 werfen. Haben Sie die richtige Lösung gefunden?

Wenn wir dieses Beispiel in einem standardkonformen Browser darstellen (denken Sie daran, dass sich der Code noch nicht für Internet Explorer eignet!) und auf das Bild klicken, sehen wir die Darstellung in Abb. 5.3.

Der Grund für die Reihenfolge der Ausgabe sollte klar sein. Der für das Element `level1` angelegte Handler ist ein Aufzeichnungshandler, die übrigen sind Aufstiegshandler. Der Handler für `level1` wird während der Aufzeichnungsphase ausgelöst und zeigt seine Ausgabe, der Handler für das Element `<img>` wird während der Zielphase ausgelöst, der Handler für `level2` schließlich während der Aufstiegsphase.

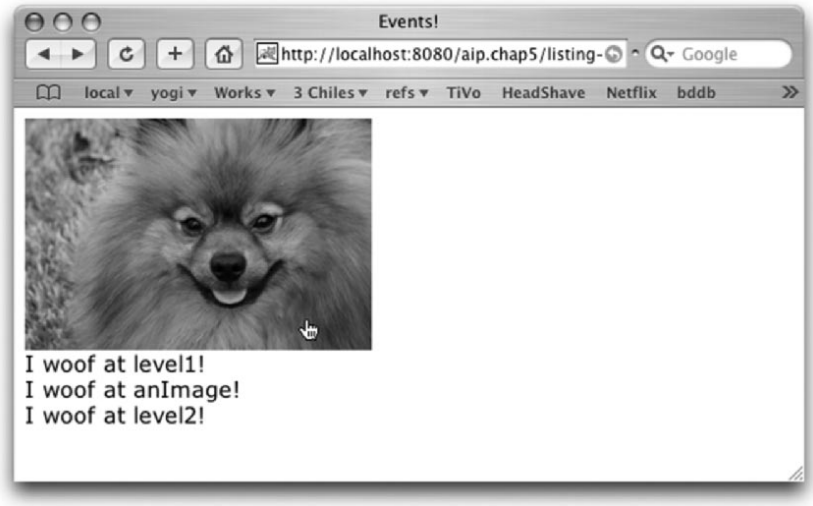

Abbildung 5.3: Ergebnis der Aufzeichnungs- und der Aufstiegsphase

Internet Explorer unterstützt nur die Ziel- und die Aufstiegsphase, nicht die Aufzeichnungsphase. Um dieses Beispiel an IE anzupassen, müssen wir anstelle von `addEventListener()` die Methode `attachEvent()` aufrufen und außerdem die Ereignishandlerfunktion ändern. Leider gibt es in der von Internet Explorer bereitgestellten Klasse `Event` keine Eigenschaft, die `currentTarget` entspricht.

Wenn IE der Zielbrowser ist und ein Verweis auf das aktuelle Zielelement der Aufstiegsphase unbedingt erforderlich ist, sind Sie auf einen Trick angewiesen, um dem Ereignishandler diesen Verweis zu liefern. Eine mögliche Taktik wäre die Verwendung des Prototype-Mechanismus `bind()`, um zu erzwingen, dass das Funktionskontextobjekt (der Verweis `this`) für den Ereignishandler das Element ist, für das der Handler angelegt wird:

```
Event.observe('someId','click',someHandler.bind($('someId')));
```

Dann können wir folgende Zeile in den Handler einfügen:

```
if (!event.currentTarget) event.currentTarget = this;
```

Damit werden Umgebungen erkannt, in denen `currentTarget` nicht definiert ist, und der Verweis auf das Kontextobjekt in die `Event`-Instanz verlegt, damit er im restlichen Handlercode browserunabhängig verwendet werden kann. Dies ist vielleicht ein wenig umständlich, aber sinnvoll, wenn diese Informationen unbedingt über Browsergrenzen hinweg zur Verfügung stehen müssen.

Die Weiterleitung beenden

Gelegentlich wollen Sie verhindern, dass die Weiterleitung eines Ereignisses fortgesetzt wird, beispielsweise, wenn Sie wissen, dass Sie das Ereignis für Ihren Bedarf ausreichend behandelt haben und die Fortsetzung der Weiterleitung unerwünschte Handler starten würde.

In einem standardkonformen Browser rufen Sie dazu innerhalb des Handlers die Methode `stopPropagation()` der Klasse `Event` auf. In IE wird die Eigenschaft `cancelBubble` der `Event`-Instanz auf `true` gesetzt. Es mag merkwürdig wirken, anstelle eines Methodenaufrufs eine Eigenschaft zu setzen, um die Weiterleitung zu beenden, aber IE definiert es nun einmal so.

Die Standardaktion verhindern

Einige Ereignisse, die als *semantische Ereignisse* bezeichnet werden, lösen im Browser eine Standardaktion aus – beispielsweise, wenn ein Formular übertragen oder ein Ankerelement angeklickt wird.

Bei DOM-Level-0-Handlern kann der Wert `false` zurückgegeben werden, um diese Standardaktion abzubrechen. Bei DOM-Level-2-Handlern steht für diesen Zweck die Methode `preventDefault()` der Klasse `Event` zur Verfügung, die verhindert, dass die Standardaktion stattfindet. Damit können Sie zum Beispiel verhindern, dass ein Formular übertragen wird, wenn eine von einem Handler für ein Übertragungsereignis durchgeführte Gültigkeitsprüfung feststellt, dass ein oder mehrere Formularfelder ungültig sind. In IE wird die Eigenschaft `returnValue` der `Event`-Instanz auf `false` gesetzt, um die Ausführung der Standardaktion zu verhindern.

All diese Browserunterschiede sind mühevoll zu handhaben. Glücklicherweise denken nicht nur wir so. Die Autoren von JavaScript-Bibliotheken sind uns zu Hilfe gekommen. Sehen wir uns an, wie eine uns inzwischen vertraute JavaScript-Bibliothek die Ereignisbehandlung auf unseren Seiten weniger mühsam macht.

5.3 Prototype für die Ereignisbehandlung

Es gibt einige JavaScript-Bibliotheken, die die Definition von Ereignishandlern vereinfachen, indem sie die Unterschiede zwischen den Browsern auf eine höhere Abstraktionsebene verlagern. Prototype, das wir in den Kapiteln 3 und 4 zur Unterstützung beim Schreiben von objektorientiertem JavaScript-Code und bei Ajax-Anfragen eingeführt haben, stellt auch eine einfache, aber bequeme Abstraktion bereit, um uns bei der Ereignisbehandlung zu helfen.

Prototype definiert einen Namespace `Event`, der eine Reihe nützlicher Methoden enthält; die beiden wichtigsten sind `observe()` und `stopObserving()`. Die erste ermöglicht das Verknüpfen eines Ereignishandlers mit einem Element, während die zweite Ereignishandler löscht.

Nehmen wir unser Beispiel aus Listing 5.6 und ändern es unter Verwendung von Prototype. Das Ergebnis sehen Sie in Listing 5.7.

Listing 5.7: Ereignishandler à la Prototype

```
<html>
  <head>
    <title>Events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {     ❶ Definiert Ereignishandler
       Event.observe('anImage', 'click', react, false);
        Event.observe('level1', 'click', react, true);
        Event.observe('level2', 'click', react, false);
      }

      function react(event) {     ❷ Deklariert Handlerfunktion
        $('info').innerHTML +=
          'I woof at ' + Event.Element(event).id + '!<br/>';
      }
    </script>
  </head>
  <body>
    <div id="level1">
      <div id="level2">
        <img src="image.jpg" id="anImage"/>
      </div>
    </div>
    <div id="info"></div>
  </body>
</html>
```

Wie viel Prototype ausmacht! Wir können nicht nur die praktische Prototype-Funktion `$()` nutzen, sondern auch unser Beispiel bei gleichzeitiger Reduzierung der erforderlichen Codemenge browserübergreifend kompatibel machen.

Im Ereignishandler `onload` ❶ haben wir unsere Handler mit der Methode `event.Observe()` browserübergreifend erstellt. Für W3C-kompatible Browser können wir immer noch festlegen, ob es sich um Aufzeichnungs- oder Aufstiegsbrowser handeln soll. Unter IE wird diese Unterscheidung einfach ignoriert.

In unserer Ereignishandlerfunktion ❷ haben wir mit Hilfe der Methode `Event.Element()` ohne Berücksichtigung des Browsers einen Verweis auf das Zielelement abgerufen.

Beachten Sie, dass Prototype die Unterschiede zwischen den Browsern hinsichtlich der Ereignisbehandlung nicht hundertprozentig beseitigt. Wollen wir zum Beispiel den Wert der Eigenschaft `currentTarget` ermitteln, müssen wir direkt vorgehen und

einen solchen Verweis vermeiden, wenn wir mit IE arbeiten. Prototype vereinfacht jedoch einen großen Teil der häufigsten Anforderungen an die Ereignisbehandlung.

5.3.1 Die Ereignis-API von Prototype

Dieser Abschnitt enthält einen Schnelldurchgang durch die API für den Prototype-Namespace `Event` mit einer Beschreibung der verfügbaren Methoden.

Die folgende Methode richtet einen Ereignishandler für den benannten Ereignistyp und das übergebene Element `Element` ein:

```
Event.observe(Element,eventType,handler,useCapture)
```

Der Parameter `useCapture` kann weggelassen werden und ist standardmäßig auf `false` gesetzt. In IE wird er ignoriert.

Die nächste Methode löscht einen Ereignishandler:

```
Event.stopObserving(Element,eventType,handler,useCapture)
```

Die Parameter sollten mit denjenigen identisch sein, die bei der Einrichtung des betreffenden Handlers verwendet wurden.

Die folgende Methode löscht alle mit `observe()` eingerichteten Handler und gibt alle Verweise frei, um sie für die Garbage Collection verfügbar zu machen. Dies ist besonders für IE wichtig, weil dort ein schweres Speicherproblem besteht, was die Ereignisbehandlung betrifft. Die beste Nachricht lautet, dass Prototype diese Methode unter IE automatisch aufruft, wenn eine Seite geschlossen wird:

```
Event.unloadCache()
```

Die folgende Methode gibt das Zielelement des übergebenen Ereignisses zurück:

```
Event.Element(event)
```

Die folgende Methode gibt den nächsten Vorfahren des Zielelements des übergebenen Ereignisses mit dem übergebenen Tagnamen zurück:

```
Event.findElement(event,tagName)
```

Damit können Sie zum Beispiel das nächste übergeordnete `<div>`-Element des Zielelements finden, indem Sie den String »div« als Parameter `tagName` übergeben.

Die beiden folgenden Methoden geben die horizontale bzw. vertikale Position eines Mausereignisses relativ zur Seite zurück:

```
Event.pointerX(event)
Event.pointerY(event)
```

Die folgende Methode gibt `true` zurück, wenn ein Mausereignis das Ergebnis eines Klicks mit der primären Maustaste war:

```
Event.isLeftClick(event)
```

Die folgende Methode beendet schließlich die Weiterleitung des Ereignisses *und* eine mit dem Ereignis verknüpfte Standardaktion:

```
Event.stop(event)
```

Jaaa! Das sollte den Code für Ereignisse um einiges einfacher machen. Wenden wir unsere Aufmerksamkeit nun den verschiedenen Ereignistypen zu, mit denen wir uns häufig befassen müssen.

5.4 Ereignistypen

Wenn es um Webanwendungen geht, wissen wir, dass die meisten für uns interessanten Ereignisse Ergebnis der Interaktion des Benutzers mit der Anwendung mittels Maus oder Tastatur sind. Diese Ereignisse werden in der DOM-Elementstruktur als Reaktion auf Benutzeraktionen ausgelöst, beispielsweise das Laden einer Seite, ein Klick auf eine Schaltfläche, das Bewegen der Maus, das Ziehen der Maus, Eingaben auf der Tastatur oder eine Aktion, die zum Schließen der Seite führt. Wie wir gesehen haben, können wir dafür Ereignishandler schreiben, damit die Anwendung auf diese Aktionen reagieren kann. In diesem Abschnitt sehen wir uns die am häufigsten behandelten Ereignistypen etwas genauer an, und zwar zuerst die Mausereignisse.

5.4.1 Mausereignisse

In einer Webanwendung werden am häufigsten folgende Mausereignisse behandelt: `mouseup`, `mousedown`, `click`, `dblclick` und `mousemove`. Wenn ein Benutzer auf ein Element klickt, werden drei Ereignisse ausgelöst: `mouseup`, `mousedown` und `click`. Betrachten wir dies zunächst anhand des Codes in Listing 5.8.

Listing 5.8: Mausereignisse bei einem Einzelklick

```
<html>
  <head>
    <title>Mouse events!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {     ❶ Legt Mausereignishandler an
        Event.observe('anImage', 'click', react);
        Event.observe('anImage', 'mousedown', react);
        Event.observe('anImage', 'mouseup', react);
      }
```

```
      function react(event) {    ❷ Gibt Informationen über das Ereignis aus
        $('info').innerHTML +=
          'I bark for ' + event.type +
          ' at (' + Event.pointerX(event) + ','+
          Event.pointerY(event) + ')!<br/>';
      }
    </script>
  </head>
  <body>
    <img src="image.jpg" id="anImage"/>
    <div id="info"></div>
  </body>
</html>
```

In diesem Code legen wir Ereignishandler für die Ereignisse `click`, `mouseup` und `mousedown` und das Element `<img>` an ❶. Bei einem Klick auf das Bild untersucht die Ereignishandlerfunktion ❷ die Instanz `event` und gibt den Ereignistyp und die Mauskoordinaten beim Klick relativ zur Seite aus. Im Browser sehen wir, was in Abb. 5.4 dargestellt ist.

Abbildung 5.4: Reaktion auf Mausereignisse

Daraus ergibt sich, dass beim Klick zuerst das Ereignis `mousedown`, dann `mouseup` und schließlich `click` ausgelöst wurde. Fügen Sie zur Übung Ereignishandler für `mousemove` und `dblclick` hinzu und beobachten Sie, wie diese in Relation zu den anderen Ereignissen übertragen werden.

5.4.2 Tastaturereignisse

Am häufigsten werden die Tastaturereignisse `keyup`, `keydown`, `blur` und `focus` behandelt. Die Ereignisse `keyup` und `keydown` entsprechen den Ereignissen `mouseup` und `mousedown`; das erste wird beim Drücken einer Taste ausgelöst, das zweite beim Loslassen.

Die Ereignisse `focus` und `blur` werden ausgelöst, wenn ein DOM-Element den Fokus erlangt bzw. verliert. Auf jeder geladenen Seite kann jeweils nur ein Element den Fokus haben. Er kann per Programm oder durch Benutzeraktion geändert werden. Wenn der Benutzer ein Feld verlässt, wird das Ereignis `blur` ausgelöst, auf das das Ereignis `focus` des nächsten Elements folgt, das den Fokus erlangt. Außerdem kann der Benutzer den Fokus durch einen Klick auf ein fokusfähiges Element ändern.

Sehen wir uns ein Beispiel für die Funktionsweise der beiden letztgenannten Ereignisse an. Betrachten Sie den Code in Listing 5.9:

Listing 5.9: blur und focus, blur und focus, ...

```
<html>
<head>
    <title>Blur and Focus</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {     ❶ Legt Handler beim Laden der Seite an
        Event.observe('nameField', 'blur', react);
        Event.observe('nameField', 'focus', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        $('nameField').focus();    ❷ Weist dem ersten Feld den Fokus zu
      }

      function react(event) {    ❸ Handhabt blur- und focus-Ereignisse
        $('info').innerHTML +=
          Event.Element(event).id + ' ' +
          event.type + '<br/>';
      }
    </script>
  </head>
  <body>
    <form name="infoForm">    ❹ Enthält fokusfähige Elemente
      <div>
        <label>Dog's name:</label>
        <input type="text" id="nameField"/>
      </div>
      <div>
```

```
        <label>Breed:</label>
        <input type="text" id="breedField"/>
      </div>
      <div>
        <label>Date of birth:</label>
        <input type="text" id="dobField"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Die Struktur dieses Beispiels ähnelt den bisher betrachteten. Wir haben jedoch einige wichtige Änderungen vorgenommen, um das Hauptaugenmerk von Maus- (in erster Linie `click`) auf Tastaturereignisse zu verlagern.

In den Rumpf der Seite wurde ein `<form>`-Element eingefügt ❹, in dem wir drei Textfelder definiert haben. Im Ereignishandler `onload` ❶ richten wir für jedes davon einen Ereignishandler `focus` und einen Ereignishandler `blur` ein. Der Deutlichkeit halber haben wir sie einzeln hinzugefügt. Zur Übung: Wie schreiben Sie den Code so um, dass alle Textfelder des Formulars mit den Ereignishandlern versehen werden, ohne sie einzeln aufzuführen?

Außerdem weisen wir am Schluss des Handlers `onload` dem ersten Feld des Formulars skriptgesteuert den Fokus zu ❷. Dies ist wichtig (und außerdem nett), weil es uns zeigt, dass der Handler `focus` für dieses erste Feld beim Laden der Seite ausgelöst wird. Dies macht deutlich, dass das Ereignis `focus` ausgelöst wird, wenn der Fokus entweder skriptgesteuert oder durch Benutzeraktivität zugewiesen wird.

Das gilt nicht für alle Ereignisse. Das Ereignis `submit` für ein Formularelement wird beispielsweise nicht gestartet, wenn ein Formular skriptgesteuert übertragen wird.

Außerdem haben wir unsere Ereignishandlerfunktion `react()` ❸ ein wenig verändert, so dass sie den Namen des Zielelements und den Ereignistyp ausgibt.

Wenn diese Seite zum ersten Mal in den Browser geladen wird, sehen wir das, was im oberen Teil von Abb. 5.5 dargestellt ist.

Wie Sie erkennen, ist der Ereignishandler `focus` bereits aufgerufen worden, weil wir dem Element `nameField` im Ereignishandler `onload` den Fokus zugewiesen haben.

Nachdem wir einige Daten eingetragen haben und zum Element `dobField` gewechselt sind, können wir sehen, dass beim Verlassen eines Feldes der Ereignishandler `blur` für das betreffende Feld aufgerufen und der Ereignishandler `focus` ausgelöst wird, sobald das nächste Element in der Reihenfolge den Fokus erhält. (Erheblich mehr über die Reihenfolge der Registerkarten erfahren Sie in Kapitel 10.)

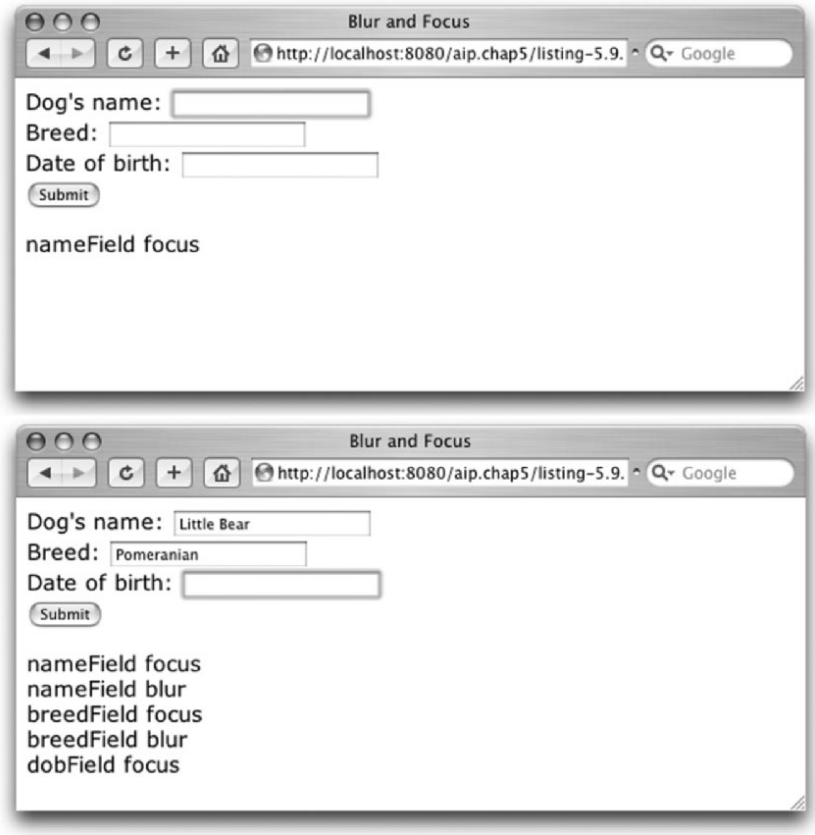

Abbildung 5.5: Fokuszuweisung und -verlust

Kopieren Sie den Beispielcode aus Listing 5.9 und fügen Sie den Textfeldern Ereignishandler für die übrigen Tastaturereignisse hinzu. Beobachten Sie, wie sie beim Eingeben von Werten in die Felder ausgelöst werden.

5.4.3 Das Änderungsereignis

Wir haben gesehen, wie wir uns mit Hilfe des Ereignishandlers `blur` benachrichtigen lassen können, wenn der Benutzer ein Feld verlässt. Es wäre ebenfalls sinnvoll zu wissen, ob sich der Wert eines DOM-Elements geändert hat, wenn es den Fokus verliert – beispielsweise, wenn wir für ein Feld nur bei geänderten Daten eine Gültigkeitsprüfung durchführen wollen anstatt bei jedem Fokusverlust. Für bestimmte Elementtypen, zum Beispiel `text`, `textarea`, `select` und `file`, löst das DOM ein Änderungsereignis aus, wenn der Fokus verloren geht und sich der Elementinhalt geändert hat, während das Element den Fokus hatte.

Um dies in Aktion zu sehen, fügen wir den Textfeldelementen unseres vorherigen Beispiels Handler für Änderungsereignisse hinzu. Das Ergebnis sehen Sie in Listing 5.10, in dem die Änderungen gegenüber Listing 5.9 durch Fettschrift hervorgehoben sind.

Listing 5.10: Sehen Sie, was sich geändert hat

```
<html>
  <head>
    <title>Ch-ch-changes</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('nameField', 'blur', react);
        Event.observe('nameField', 'focus', react);
        Event.observe('nameField', 'change', react);
        Event.observe('breedField', 'blur', react);
        Event.observe('breedField', 'focus', react);
        Event.observe('breedField', 'change', react);
        Event.observe('dobField', 'blur', react);
        Event.observe('dobField', 'focus', react);
        Event.observe('dobField', 'change', react);
        $('nameField').focus();
      }

      function react(event) {
        $('info').innerHTML +=
          Event.Element(event).id + ' ' +
          event.type + '<br/>';
      }
    </script>
  </head>
  <body>
    <form name="infoForm">
      <div>
        <label>Dog's name:</label>
        <input type="text" id="nameField"/>
      </div>
      <div>
        <label>Breed:</label>
        <input type="text" id="breedField"/>
      </div>
      <div>
        <label>Date of birth:</label>
        <input type="text" id="dobField"/>
      </div>
      <div>
```

```
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Mit nur geringfügigen Änderungen des HTML-Dokuments haben wir die Fähigkeit gewonnen, uns bei Änderungen in den Textfeldern unseres Formulars benachrichtigen zu lassen.

Wenn wir diese Seite in den Browser laden, Text in das erste Feld eingeben, zum zweiten Feld wechseln und dann zum dritten, ohne etwas in das zweite einzugeben, sehen wir etwa das, was Abb. 5.6 zeigt.

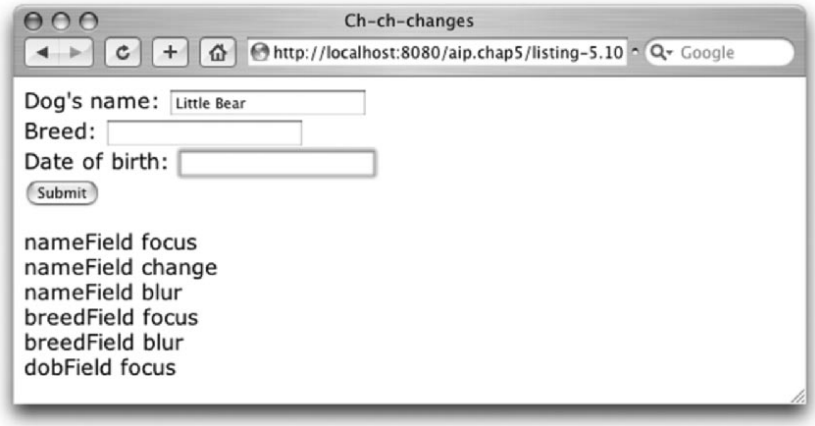

Abbildung 5.6: Was hat sich geändert?

Wie Sie feststellen, wurde direkt vor dem Ereignis `blur` für das Namensfeld, dessen Wert durch Benutzereingabe geändert wurde, ein Änderungsereignis ausgelöst, jedoch nicht für das Feld BREED, das gleich geblieben ist.

5.4.4 Seitenereignisse

Bisher ging es um Ereignisse, die ausgelöst werden, wenn ein Benutzer mit den Elementen einer *geladenen* Seite interagiert. Er kann jedoch auch Ereignisse für Aktivitäten auf Seitenebene auslösen. Diese *Seitenereignisse* finden statt, wenn ein Dokument geladen, geschlossen, in der Größe verändert oder der Fensterinhalt verschoben wird. Obwohl diese Ereignisse etwas Besonderes zu sein scheinen, können wir sie genauso erfassen wie andere, indem wir Ereignishandler für das Element `<body>` der Seite bereitstellen oder mit Hilfe des `window`-Objekts zuweisen.

In allen bisherigen Beispielen dieses Kapitels haben wir bereits das Ereignis `load` in Aktion gesehen: Wir haben es zum Deklarieren der übrigen Ereignishandler benutzt, die wir vorführen wollten. Erweitern wir die Mischung nun um Beispiele für die Ereignisse `unload` und `onbeforeunload` (siehe Listing 5.11).

Listing 5.11: Seitenereignisse

```
<html>
  <head>
    <title>Page Events</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {    ❶ Benachrichtigt, dass Seite geladen wird
        alert('Loaded!');
        window.onunload = function() {    ❷ Benachrichtigt, dass Seite entladen wird
          alert('Unloaded!');
        }
        window.onbeforeunload =    ❸ Bietet Wahlmöglichkeit
          function() {
            return 'Leaving so soon?';
          }
        }
      }
    </script>
  </head>
  <body>
    <a href="listing-5.11.html">Do it again!</a>
  </body>
</html>
```

Da wir vorhaben, die ganze Seite zu laden und zu schließen, sind Ausgaben auf der Seite zur Beobachtung des Geschehens nicht besonders sinnvoll, so dass wir wieder auf Meldungsfelder ausgewichen sind. Im Ereignishandler `onload` geben wir eine Meldung aus, wenn die Seite geladen wird ❶ und fahren dann damit fort, dass wir Ereignishandler für die Ereignisse `unload` und `beforeunload` einrichten.

Im Ereignishandler `onunload` ❷ geben wir einfach eine weitere Meldung aus, die besagt, dass das Ereignis ausgelöst wurde. Der Handler `onbeforeunload` ist dagegen ein wenig interessanter.

Im Handler `onunload` können wir wenig mehr tun als auf den Umstand zu reagieren, dass die Seite geschlossen wird, aber im Handler `onbeforeunload` haben wir tatsächlich Einfluss darauf, ob die Seite geschlossen wird oder nicht. Wenn wie in unserem Handler `onbeforeunload` ein Wert zurückgegeben wird ❸, zeigt der Browser ein Dialogfeld an, das den Benutzer fragt, ob die Seite wirklich geschlossen werden soll. Es enthält den vom Handler zurückgegebenen Wert als Bestandteil des Textes.

Laden wir dieses Beispiel in den Browser, erhalten wir die störende Meldung, dass die Seite geladen wurde. Wenn wir auf den Link auf der Seite klicken, der einfach dieselbe Seite erneut anzeigen soll, sehen wir, dass der Browser unseren Ereignishandler `onbeforeunload` auslöst und dass als Ergebnis des Werts, den wir von diesem Handler zurückbekommen haben, das Dialogfeld in Abb. 5.7 erscheint.

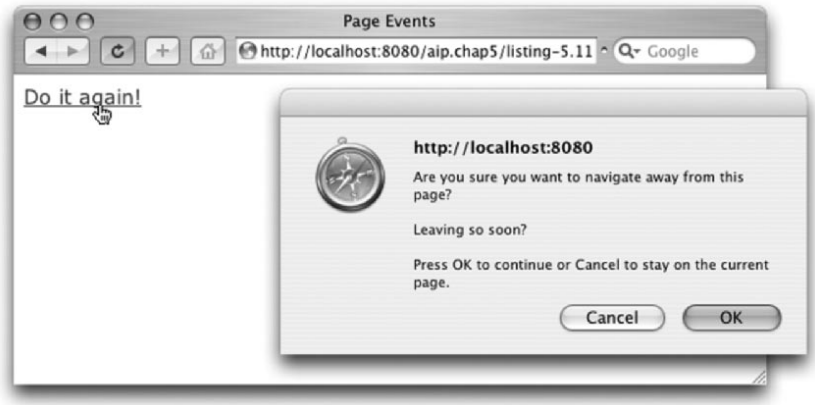

Abbildung 5.7: Auf ein Wort, bevor du gehst

Es braucht nicht viel Vorstellungskraft, um zu erkennen, dass diese Technik ganz sinnvoll sein kann, damit die Benutzer keine Daten verlieren, wenn sie versuchen, eine Seite zu verlassen, bevor sie ihre Tätigkeit abgeschlossen haben. Klickt der Benutzer auf die Schaltfläche ABBRECHEN, wird die Seitennavigation abgebrochen, und der Schließvorgang findet nicht statt. Klickt er auf OK, wird der Vorgang fortgesetzt und die Meldung eingeblendet, dass direkt vor dem erneuten Laden der Seite der Ereignishandler `unload` aufgerufen wurde.

Eine Anmerkung zur Verwendung des `load`-Ereignisses: Häufig sieht man Seiten, auf denen gegen Ende ein `<script>`-Element platziert ist, das beim Laden der Seite Code ausführen soll. Der Unterschied zwischen dieser Taktik und der Implementierung von `load` besteht darin, dass das Ereignis `load` garantiert erst ausgelöst wird, wenn der Ladevorgang *abgeschlossen* ist, um externe Elemente wie Skriptdateien, CSS-Stylesheets und Bilder einzubinden.

Damit ist unser Überblick über die Ereignisbehandlung und die Erörterung einiger häufig vorkommender Ereignistypen abgeschlossen. Natürlich haben wir nicht alle Ereignisse untersucht, die innerhalb einer Seite ausgelöst werden können – damit lassen sich mehrere Kapitel füllen –, aber die hier gegebenen Informationen reichen mit Sicherheit aus, um zu verstehen, wie Ereignisbehandlung abläuft und wie die Ereignistypen zu handhaben sind, die in modernen Webanwendungen am häufigsten verwendet werden.

5.5 Ereignisse im Einsatz

Für die Ausführung der Beispiele in diesem Abschnitt sind serverseitige Ressourcen erforderlich. Um unseren Lesern möglichst viel Mühen zu ersparen, ist der Beispielcode für dieses Kapitel unter der Adresse *www.manning.com/crane2* bereits als vollständige und lauffähige Webanwendung eingerichtet.

Wenn auf Ihrem System bereits ein Servlet-Container in Betrieb ist, legen Sie einfach einen neuen Anwendungskontext mit dem Namen `aip.chap5` an, der auf den Ordner `chap5` des heruntergeladenen Codes als Dokumentbasis zeigt.

Haben Sie noch kein Servlet-Modul, brauchen Sie auch nicht in Panik zu geraten. Ein PDF-Dokument im Ordner `chap4` des Codes leitet Sie an, Tomcat herunterzuladen und zu konfigurieren und zeigt Ihnen auch, wie Sie Anwendungskontexte einrichten.

Achten Sie beim Öffnen dieser Beispiele im Browser darauf, dass Sie die HTML-Seiten nicht lediglich als Dateien öffnen, sondern über den Webserver aufrufen. Um das Beispiel in Listing 5.12 zu laden, benutzen Sie beispielsweise folgende Adresse:

http://localhost:8080/aip.chap5/listing-5.12.html

Dabei wird natürlich vorausgesetzt, dass Sie den Servlet-Container auf dem Standardport 8080 betreiben. Benutzen Sie einen anderen, müssen Sie die URL zwangsläufig anpassen.

5.5.1 Textfelder auf dem Server validieren

Mit dem Wissen, wie Sie die Ereignishandler `change` und `blur` mit DOM-Elementen verknüpfen, ist die Validierung von Eingabeelementen auf dem Client mit Hilfe dieser Handler recht einfach. Einfache Prüfungen auf der Clientseite lassen sich leicht durchführen, aber gelegentlich schreiben Geschäftserfordernisse vor, die Daten unter Verwendung von Kenntnissen zu validieren, die nur auf dem Server zur Verfügung stehen. Der Grund kann darin liegen, dass die Validierung zu komplex für die Erledigung mit JavaScript ist oder dass die erforderlichen Daten für die Validierung zu umfangreich sind, um sie zur Verwendung auf dem Client an die Seite zu senden.

Eine gängige Vorgehensweise klassischer Webanwendungen sieht so aus, dass die einfache Validierung auf der Seite erfolgt, die komplexeren Prüfungen jedoch erst bei der Übertragung des Formulars. Dank Ajax brauchen wir den Benutzer jedoch nicht mehr durch dieses ziemlich schizophrene Validierungsverfahren zu führen. Um Validierungen mit Serverunterstützung im laufenden Betrieb durchzuführen, richten wir eine Anfrage an den Server, wenn auf dem Client ein entsprechendes Ereignis eintritt. Der Server validiert dann die Daten und sendet dem Client eine entsprechende Meldung.

Wir haben alle Informationen zur Lösung dieses Problems. Wir wissen, dass wir ein Ereignis mit einem Textfeld verknüpfen können, um Änderungen festzustellen, und dass wir mit Ajax dieses Ereignis benutzen können, um eine Anfrage an den Server zu senden. Die serverseitige Ressource, mit der eine solche Anfrage Kontakt aufnimmt, kann die Daten validieren und eine Fehlermeldung zurücksenden, wenn sie sich als ungültig erweisen.

Beachten Sie, dass der Zweck des Beispiels in diesem Abschnitt nicht darin besteht, ein ausgefeiltes oder raffiniertes Framework für die Validierung zu präsentieren, sondern praktischen Einsatz der Ereignisbehandlung vorzuführen. Die Validierung kommt im weiteren Verlauf des Buches noch an die Reihe, in Kapitel 6 und dann noch einmal in Kapitel 10.

Problem

Wir sollen Textfelder mit Hilfe einer serverseitigen Ressource validieren, wenn sich ihr Wert ändert.

Lösung

Wir haben bereits gesehen, wie man ein Eingabetextelement mit Ereignishandlern ausstattet, was auch bei dieser Lösung der Fall ist. Die Frage lautet: Fangen wir `blur`- oder `change`-Ereignisse ab?

Die Antwort hängt vom Charakter der Daten und den durchzuführenden Validierungen ab. Da wir vorhaben, bei jeder Validierung mit Serverassistenz eine Reise zum Server zu machen, wollen wir sicherstellen, dass wir nicht mehr Anfragen starten als erforderlich.

Wenn wir wissen, dass die Ausgangsdaten gültig sind, können wir uns auf Änderungsereignisse beschränken. Schließlich brauchen wir keine Daten zu prüfen, von denen wir wissen, das sie in Ordnung sind. In dem häufigeren Fall, dass Felder zu Anfang unbekannte Daten enthalten können (oder sogar leer sind), müssen wir wahrscheinlich `blur`-Ereignisse abfangen, damit das Feld bei jedem Besuch validiert werden kann.

Einen Ereignishandler für das zu prüfende Feld zu erstellen ist ganz einfach:

```
Event.observe('fieldId','blur',validationFunction);
```

Listing 5.12 enthält eine Seite mit einem kurzen Formular, das aus Feldern für eine US-Adresse, nämlich Stadt, Staat und Postleitzahl, besteht. Unsere Geschäftsregeln schreiben vor, dass Postleitzahl und Adresse zusammenpassen müssen. Dazu muss eine serverseitige API herangezogen werden, die der US-Postdienst zur Verfügung stellt und die im Servercode befragt werden muss. Sehen wir uns an, wie wir das auf der Seite bewerkstelligen.

Listing 5.12: Validieren der Postleitzahl

```
<html>
  <head>
    <title>I Need Validation</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {    ❶ Richtet die Ereignisbehandlung ein
        Event.observe('zipCodeField','blur',validateZipCode);
        $('addressField').focus();
      }

      function validateZipCode(event) {    ❷ Sendet eine Validierungsanfrage
        new Ajax.Request(
          '/aip.chap5/validateZipCode',
          {
            method: 'get',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              if (transport.responseText.length != 0)
                alert(transport.responseText);
            }
          }
        );
      }
    </script>
  </head>

  <body>
    <form id="infoForm">    ❸ Richtet ein Formular für die Dateneingabe ein
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Diese Seite erledigt drei wichtige Aktivitäten: Sie richtet die Ereignisbehandlung ein ❶, reagiert auf das `blur`-Ereignis, indem sie die Validierungsanfrage an die serverseitige Ressource sendet ❷, und richtet das Formular für die Dateneingabe zum Ausfüllen durch den Benutzer ein ❸.

Im Ereignishandler `onload` für die Seite ❶ richten wir den Handler für das `blur`-Ereignis ein, so dass die Funktion `validateZipCode()` aufgerufen wird, sobald der Benutzer das Postleitzahlenfeld verlässt. Diese Funktion ❷ sendet mit Assistenz von Prototype eine Ajax-Anfrage an die serverseitige Ressource `validateZipCode`. Wie Sie gleich sehen werden, handelt es sich um ein Java-Servlet, das einfach ein bisschen trickst, um einen Validierungsvorgang für die Postleitzahl zu emulieren.

Dieser Ressource übergeben wir mit Hilfe der nützlichen Methode `serialize()` die Felder unseres Formulars, die Prototype bequem in unser `<form>`-Element einfügt.

Die Validierungsressource auf dem Server ist so definiert, dass sie eine leere Antwort zurückgibt, wenn alles in Ordnung ist, bei gescheiterter Validierung dagegen eine Fehlermeldung. Im Ereignishandler `onSuccess` für die Validierung testen wir daher den Text der Antwort und geben eine einfache Meldung aus, wenn das Feld die Prüfung nicht bestanden hat. Denken Sie daran, dass ausgefeiltere Validierungsverfahren erst in späteren Kapiteln zur Sprache kommen.

Laden Sie die Seite in einen Browser und füllen Sie die Felder aus. (Achten Sie darauf, nicht das Dateimenü, sondern die URL zu benutzen.) Beachten Sie, dass beim Verlassen des Postleitzahlenfeldes ein Feld mit der Meldung erscheint, dass die Validierung gescheitert ist, wie Sie es in Abb. 5.8 sehen.

Abbildung 5.8: Die Postleitzahl ist ungültig!

Sie werden feststellen, dass jede von Ihnen eingegebene Postleitzahl zu einer Fehlermeldung führt, wenn Sie nicht zufällig den richtigen Wert `01826` erraten haben, weil das Validierungs-Servlet auf dem Server in Wirklichkeit natürlich keine Verbindung

mit der Datenbank des US-Postdienstes aufnimmt, um eine echte Validierung durchzuführen. Den Servlet-Code, der einen Validierungsvorgang nachahmt, sehen Sie in Listing 5.13.

Listing 5.13: Wir mogeln uns durch die Postleitzahlvalidierung.

```
package org.aip.chap5;

import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Blendwerk-Validierungs-Servlet für Listing 5.12. Die
 * Postleitzahl muss ausgefüllt und gleich "01826" sein,
 * um als gültig zu gelten.
 */
public class ZipCodeValidatorServlet extends HttpServlet {

  protected void doGet(HttpServletRequest request,
                       HttpServletResponse response)
       throws IOException {
    StringBuilder result = new StringBuilder ();
    String zipCodeValue = request.getParameter("zipCode");
    if (zipCodeValue.length() == 0) {
      result.append("The zip code field cannot be blank");
    }
    else if (!zipCodeValue.equals("01826")) {
      result
        .append("The zip code value of ")
        .append(zipCodeValue)
        .append(" does not match the specified street address");
    }
    response.getWriter().write(result.toString());
  }
}
```

Es gibt hier wirklich nicht viel zu kommentieren, außer dass bei einer echten Validierung sämtliche Formularfelder abgefragt und eine vom US-Postdienst bereitgestellte API benutzt würde. Da dies aber nicht der Schwerpunkt dieses Beispiels (und schon gar nicht des Buches) ist, verwenden wir hier lediglich eine Scheinressource, die es ermöglicht, den Clientcode in Aktion zu beobachten.

Diskussion

In diesem Abschnitt haben wir eine Hybridmethode von aus Client angestoßener Validierung und Serverassistenz gesehen, die uns in die Lage versetzt, dem Benutzer

unabhängig davon, ob die Validierung Serverressourcen benötigt, unmittelbare Rückmeldungen über die von ihm eingegebenen Daten zu geben.

Mit Hilfe des `blur`-Ereignisses haben wir festgestellt, wann der Benutzer ein Feld verlassen hat, um die Prüfung zu starten. Gibt es vielleicht eine elegantere Lösung? Nach der ersten Prüfung der Daten besteht keine Notwendigkeit mehr, den Zusatzaufwand für eine weitere Reise zum Server auf sich zu nehmen, solange die Daten unverändert bleiben. Wie ändern Sie den Code so, dass die Prüfung mit Hilfe des Servers nur gestartet wird, wenn nicht bekannt ist, ob die Daten gültig sind?

Dieser Hybridansatz des gemeinsamen Einsatzes von Client und Server zur Validierung stellt eine leistungsstarke Ergänzung unserer Toolbox für Webanwendungen dar. Direkte Validierung dieser Art kann eine Menge Frustration beim Benutzer vermeiden, die sich daraus ergibt, dass er *nach* der Übertragung des Formulars von Problemen mit den eingegebenen Daten erfährt. Deshalb sollten Sie um jeden Preis eine solche Validierung implementieren. Sie können sich jedoch niemals darauf *verlassen*!

Unser clientseitiger Code steht jedem Besucher unserer Seiten ohne Weiteres zur Verfügung, und böswillige Menschen werden merken, dass sie mit Hilfe von Reverse-Engineering leicht falsche Daten übertragen können, indem sie jedes Validierungskonstrukt auf der Clientseite ohne Rücksicht darauf umgehen, wie klug es angelegt ist. Den Code für die vom Client ausgehende und vom Server unterstützte Validierung (wie den in diesem Beispiel) können Sie auch für die endgültigen Prüfungen bei der Übertragung benutzen.

Manchmal möchten wir möglicherweise bei der Übertragung von Formularen an den Server auf den zusätzlichen Aufwand verzichten, die Seite vollständig neu zu laden. Sehen wir uns das als Nächstes an.

5.5.2 Formularelemente ohne Seitenübertragung senden

Die weitaus meisten Webseiten, die Eingaben akzeptieren, sind heute in der klassischen Technik verfasst, dass sie ein Formular an den Server übertragen, wenn die Dateneingabe abgeschlossen ist. Dies zieht eine vollständige Aktualisierung der Seite nach sich, was im Kontext der reichhaltigen Webanwendungen, die wir jetzt mit Ajax übermitteln können, unerwünscht sein kann.

Problem

Wir wollen ein Formular an eine Serverressource senden, ohne die Seite vollständig neu zu laden.

Lösung

Wie sich herausgestellt hat, ist die Lösung ziemlich trivial. Eigentlich haben wir diese Aufgabe im vorherigen Beispiel bereits weitgehend erledigt. Um das Formular zu »übertragen«, benutzen wir dieselbe Technik, die wir im Beispiel verwendet haben, um Formularelemente zur Validierung an den Server zu senden.

So trivial und bekannt diese Lösung auch sein mag, machen ein paar Nuancen dieses Problem jedoch betrachtenswert. Wir nehmen den Beispielcode, löschen die Validierung (damit wir uns auf die Übertragung konzentrieren können) und fangen den Vorgang der Formularübertragung ab, um das Formular nicht durch normale Übertragung, sondern von Ajax gesteuert an den Server zu senden. Die Ergebnisse sehen Sie in Listing 5.14.

Listing 5.14: Abfangen des Übertragungsvorgangs

```
<html>
  <head>
    <title>Submit!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {    ❶ Erstellt einen Ereignishandler für Übertragung
        Event.observe('infoForm','submit',submitMe);
        $('addressField').focus();
      }

      function submitMe(event) {    ❷ Überträgt das Formular unter Kontrolle von Ajax
        new Ajax.Request(
          '/aip.chap5/handleSubmission',
          {
            method: 'post',
            parameters: $('infoForm').serialize(true),
            onSuccess: function (transport) {
              $('info').innerHTML = transport.responseText;
            }
          }
        );
        Event.stop(event);
      }
    </script>
  </head>

  <body>
    <form id="infoForm"    ❸ Weist die normale Übertragungsaktion zu
          action="/aip.chap5/shouldNotActivate">
      <div>
        <label>Address:</label>
        <input type="text" id="addressField" name="address"/>
      </div>
      <div>
        <label>City:</label>
        <input type="text" id="cityField" name="city"/>
        <label>State:</label>
        <input type="text" id="stateField" name="state"/>
```

```
        <label>Zip Code:</label>
        <input type="text" id="zipCodeField" name="zipCode"/>
      </div>
      <div>
        <input type="submit" id="submitButton"/>
      </div>
    </form>
    <div id="info"></div>
  </body>
</html>
```

Die Änderungen an der Seite sind gering, aber schwerwiegend. Zunächst haben wir dem Formular im Handler `onload` des Fensters einen Handler für das `submit`-Ereignis hinzugefügt ❶, der bewirkt, dass die Funktion `submitMe()` aufgerufen wird, wenn das Formular übertragen wird ❷.

Mit dieser Funktion befassen wir uns gleich, aber sehen Sie sich zunächst an, wie wir das Formularelement geändert haben ❸. Wir haben ein Attribut `action` hinzugefügt, das eine nicht vorhandene Ressource auf dem Server angibt. Dadurch erfahren wir schnell, ob unser Formular jemals mit der normalen Standardaktion übertragen wird: Wenn der Server meldet, dass er die Ressource nicht finden kann, zeigt der Browser eine unmissverständliche Fehlerseite an.

Die Funktion `submitMe()`, die beim Auslösen des Übertragungsereignisses aufgerufen wird, startet eine ähnliche Ajax-Anfrage wie im vorherigen Beispiel. In diesem Fall haben wir jedoch anstelle von `'get'` die HTTP-Methode `'post'` festgelegt. Die Schwerarbeit erledigt die Prototype-Methode `serialize()`.

Die serverseitige Ressource für die Anfrage ist ein Servlet, das die Parameter der Anfrage sammelt und eine Antwort formatiert, die HTML-Code mit Namen und Werten dieser Parameter enthält. (Da seine Funktionsweise für diese Erörterung nicht relevant ist, betrachten wir es hier nicht näher. Sollten Sie neugierig sein, finden Sie den Quellcode des Servlets jedoch als Klasse `org.aip.chap5.ParameterInspectorServlet` im herunterladbaren Code.) Dieser Klassenrumpf wird auf der Seite im Element `<info>` angezeigt.

Als Letztes wird die folgende Anweisung ausgeführt:

```
Event.stop(event);
```

Diese Prototype-Methode beendet die Weiterleitung des Ereignisses und bricht die Standardaktion ab, in diesem Fall also die Übertragung des Formulars. Ohne sie würde das Formular weiter an die Ressource gesendet, die im Formularattribut `action` steht.

Diskussion

Dieses Beispiel hat zwar nicht viel Neues gebracht, aber doch einige wichtige Konzepte herausgestellt, beispielsweise die Verwendung des `submit`-Ereignisses zur Ver-

hinderung der Übertragung. Für diesen Zweck haben wir einen Ereignishandler und die Ereignismethoden von Prototype eingesetzt. Wollen Sie jedoch lediglich die Übertragung des Formulars verhindern, können Sie auch die folgende Formulardeklaration benutzen, damit ein DOM-Level-0-Handler `false` zurückgibt:

```
<form id="my Form" action="whatever" onsubmit="return false;">
```

In unserem Beispiel haben wir uns außerdem stark auf die Dienste der Prototype-Methode `serialize()` gestützt. Sie ordnet die Werte aller im Formular enthaltenen Elemente und erstellt daraus entweder einen Query-String oder einen Objekt-Hash. Da wir als Parameter für diese Methode `true` angegeben haben, gibt sie einen Objekt-Hash zurück, eine Technik, die in Prototype 1.5 bevorzugt wird.

Wenn Sie diese Seite laden, Daten eingeben und auf SUBMIT klicken (oder Enter drücken), erhalten Sie eine Darstellung wie in Abb. 5.9.

Abbildung 5.9: Übertragen, ohne zu übertragen!

Das war alles recht einfach. Aber was, wenn wir ein wenig wählerischer sein wollen?

5.5.3 Nur veränderte Elemente übertragen

Das vorstehende Beispiel hat uns gezeigt, dass wir den Vorgang der Formularübertragung steuern und die übertragenen Daten durch Ereignisbehandlung an eine Ajax-Anfrage umleiten können. Die Prototype-Methode `serialize()` hat es uns sehr einfach gemacht, sämtliche Datenelemente eines Formulars zu sammeln, um sie an den Server zu senden.

Was aber, wenn wir nun aber nicht *sämtliche* Formulardaten, sondern nur geänderte Datenelemente senden wollen? Nun, warum sollten wir die Anfrage überhaupt stellen, wenn sich gar keine Daten geändert haben? Über das Ereignis `change` der Formularelemente können wir erfahren, dass sich der Wert eines Elements geändert hat, aber wie verfolgen wir diese Informationen am günstigsten, um sie zu verwenden,

wenn der Zeitpunkt zum Senden gekommen ist? Wir können dilettantisch verfahren und die Daten in globalen Variablen ablegen. Das ist aber nicht nur wenig elegant, sondern führt auf Seiten mit mehreren Formularen auch zu ernsten Problemen und ist kein objektorientierter Ansatz.

Wir können auch raffiniert vorgehen und die Informationen im Element selbst speichern, indem wir eine benutzerdefinierte Eigenschaft folgender Art einbringen:

```
Element.hasChanged = true;
```

Wenn es Zeit ist, die Daten für die Übertragung zu sammeln, können wir so die Elemente durchgehen und schauen, bei welchen die Eigenschaft gesetzt ist.

Noch besser ist es, clever zu sein (das klingt so viel besser als faul) und Code zu nutzen, den wir bereits zur Hand haben. Listing 5.15 zeigt einen solchen Ansatz.

Listing 5.15: Nur veränderte Daten übertragen

```
<html>
  <head>
    <title>Submit, or not!</title>
    <script type="text/javascript" src="prototype-1.5.1.js">
    </script>
    <script type="text/javascript">
      window.onload = function() {
        Event.observe('infoForm','submit',submitMe);
        Event.observe('infoForm','change',
                      markChanged);
        $('addressField').focus();
      }

      function markChanged(event) {
        Event.Element(event).addClassName('changedField');
      }

      function submitMe(event) {
        var changedElements = $$('.changedField');
        if (changedElements.length > 0 ) {
          var parameters = {};
          changedElements.each(
            function(Element) {
              parameters[Element.name] = Element.value;
              Element.removeClassName('changedField');
            }
          );
          new Ajax.Request(
            '/aip.chap5/handleSubmission',
            {
```

❶ **Erstellt einen Änderungshandler für das Formular**

❷ **Markiert Zielelement als verändert**

❸ **Sammelt nur veränderte Elemente**

```
            method: 'post',
            parameters: parameters,
            onSuccess: function (transport) {
              $('info').innerHTML = transport.responseText;
            }
          }
        );
      }
      Event.stop(event);
    }
  </script>
 </head>

 <body>
  <form id="infoForm" action="/aip.chap5/shouldNotActivate">
    <div>
      <label>Address:</label>
      <input type="text" id="addressField" name="address"/>
    </div>
    <div>
      <label>City:</label>
      <input type="text" id="cityField" name="city"/>
      <label>State:</label>
      <input type="text" id="stateField" name="state"/>
      <label>Zip Code:</label>
      <input type="text" id="zipCodeField" name="zipCode"/>
    </div>
    <div>
      <input type="submit" id="submitButton"/>
    </div>
  </form>
  <div id="info"></div>
 </body>
</html>
```

In diesem Beispiel haben wir ein paar kleine, aber folgenreiche Änderungen am Code aus Listing 5.14 vorgenommen. Im Ereignishandler `onload` haben wir einen Ereignishandler `change` für das Formular erstellt ❶. Wir hätten das Formular durchgehen und für jedes einzelne Element einen Handler einfügen können, aber warum sich Mühe machen, wenn das Formular während der Aufstiegsphase die Ereignisbenachrichtigung bekommt?

Die Handlerfunktion `markChanged()`❷, die jedes Mal aufgerufen wird, wenn sich ein Formularelement verändert hat, erhält einen Verweis auf das Zielelement des Ereignisses und fügt ihm die CSS-Klasse `changedField` hinzu. Huch? Was hat CSS mit der Verfolgung von Feldänderungen zu tun? Alles wird klar, wenn wir die Änderungen an der Ereignishandlerfunktion `submitMe()` betrachten.

In dieser Funktion ❸ verwenden wir die praktische Prototype-Funktion `$$()`. Sie gibt ein Array aller Elemente zurück, die mit dem als Parameter übergebenen CSS-Selektor übereinstimmen. Da wir den String `'.changedField'` angegeben haben, wird ein Array aller Elemente zurückgegeben, die mit dem Namen dieser CSS-Klasse markiert sind.

Ist das Array leer, überspringen wir einfach den Code, der die Anfrage überträgt. Andernfalls durchlaufen wir eine Schleife durch die Elemente und erstellen dabei einen Objekt-Hash der Name-Wert-Paare, die wir aus den Arrayelementen entnehmen. Dieser Hash wird anschließend als Parametermenge für die Ajax-Anfrage verwendet.

Da die Daten übertragen wurden und nicht mehr als geändert gelten, löschen wir den CSS-Klassennamen `changedField` aus den Elementen und können wieder loslegen!

Diskussion

Dieses Beispiel baut auf dem Code in Listing 5.14 auf und soll die in der Ajax-Anfrage übertragenen Parameter auf diejenigen beschränken, deren Wert verändert wurde, bzw. die Übertragung der Anfrage ganz überspringen, wenn keine Änderungen stattgefunden haben.

Wir haben einen `change`-Ereignishandler für das Formular verwendet, um Änderungen sämtlicher Elemente abzufangen, wobei wir klugerweise die Aufstiegsphase der Ereignisweiterleitung genutzt haben. Außerdem haben wir eine geschickte Methode kennen gelernt, Elemente durch CSS-Klassennamen und die Prototype-Funktion `$$()` zur späteren Identifizierung zu markieren.

Wenn wir die Seite im Browser aufrufen, nachdem sich nur die Felder für Stadt und Staat geändert haben, sehen wir etwa das, was Abb. 5.10 zeigt.

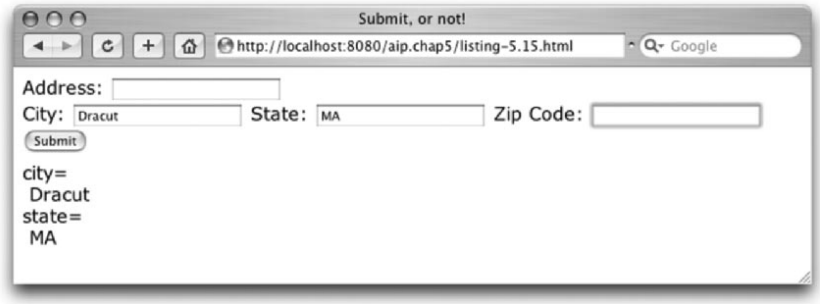

Abbildung 5.10: Nur übertragen, was zählt

5.6 Zusammenfassung

In diesem Kapitel haben wir interessante und leistungsfähige Techniken kennen gelernt, um Interaktivität in Webanwendungen zu integrieren. Wir haben die verschiedenen Verfahren betrachtet, DOM-Elementen Ereignishandler hinzuzufügen, und gesehen, wie großartig die JavaScript-Bibliothek Prototype das Verknüpfen und Schreiben von Ereignishandlern vereinfacht. Wir haben sämtliche wichtigen Ereignistypen sowie zahlreiche Codebeispiele untersucht, die vorführten, wie diese Ereignisse in unseren Webanwendungen eingesetzt werden können. Außerdem haben wir Beispiele für Validierung und Formularübertragung behandelt, was wir im nächsten Kapitel noch vertiefen werden.