11 Die Zukunft

Bisher habe ich in diesem Buch nur solche Webfunktionen beschrieben, die wenigstens in einigen Browsern relativ stabil sind oder in naher Zukunft stabil sein sollten. Jetzt sind wir aber im letzten Kapitel angelangt, und ich kann endlich einige der eher experimentellen Funktionen am Horizont ansprechen.

Überall sind Veränderungen geplant: Eine neue Revision von JavaScript mit dem Codenamen Harmony, die ursprünglich im Lauf des Jahres 2013 herauskommen sollte, wird nun für 2014 erwartet und sollte dann in den nächsten Jahren Eingang in die Browser finden; dem W3C wurden viele neue APIs vorgeschlagen, darunter auch eine zum Aufspüren von Netzwerkgeräten mittels Universal Plug and Play (UPnP) und eine zur Messung des Umgebungslichts; die Arbeit am Entwurf der HTML5.1-Spezifikation macht gute Fortschritte und viele CSS-Module streben bereits das Level 4 an. Ich könnte endlose Veränderungen aufzählen – aber ich konzentriere mich auf solche, die unsere Arbeitsweise meiner Ansicht nach am stärksten beeinflussen und gute Chancen auf eine Implementierung haben werden.

11.1 Webkomponenten

Web Components

Es ist kaum übertrieben zu sagen, dass die Web-Components-Spezifikation die radikalsten Veränderungen an HTML seit seiner Entstehung vor über 20 Jahren vorschlagen. Selbst das so stark gehypte HTML5 ist in Wahrheit nur ein kleines Versionsupdate, das nichts wirklich Neues mitbringt.

Web Components ist ein Sammelbegriff für eine Reihe von Ergänzungen zu HTML und dem DOM, die die Erstellung multifunktionaler Benutzeroberflächen für Webanwendungen unterstützen – eine Art wiederverwendbare Widget-Spezifikation. Während ich dies schreibe, existieren vier Hauptkomponenten: *Templates*, *Decorator*, *Custom Ele-*

ments und das *Shadow DOM*. Ich beschreibe ihre Funktion nachfolgend im Einzelnen – zuerst möchte ich aber zusammenfassen, was sie gemeinsam leisten können.

Eines der Hauptprobleme beim Entwickeln von Anwendungskomponenten in HTML besteht heute darin, dass ihre Bestandteile (sprich: Elemente) Teil des DOM sind. Aus diesem Grund sind sie offen für Konflikte mit CSS oder JavaScript. Das könnten Vererbungskonflikte sein, wie auf übergeordnete Elemente angewandte Regeln, die in Komponentenelemente kaskadieren, oder – umgekehrt – auf Komponentenelemente angewandte Regeln, die auf andere Elemente im DOM kaskadieren.

Ein weiteres Problem sind Namenskonflikte, wenn dieselbe Klasse oder ID unwissentlich auf verschiedenen Seiten einer Site eingesetzt wird. Für ein Element bewusst deklarierte Regeln werden so unbeabsichtigt auch auf andere Elemente angewandt. Dieses Problem findet sich häufig auf großen Sites, denen ein klares Schema bei der Namensgebung fehlt. Durch Konflikte in JavaScript kann es sich sogar noch verschlimmern, wenn Selektoren unerwünschtes Funktionsverhalten auf ein Element anwenden.

Am besten lassen sich solche Konflikte durch Abtrennung der Komponente vom Rest des DOM vermeiden; jegliche Abhängigkeit wird damit vermieden. Diese in der objektorientierten Programmierung grundlegende Technik ist als *Datenkapselung* bekannt.

Datenkapselung im HTML-DOM

Mit Web Components wird versucht, die Datenkapselung im HTML-DOM umzusetzen; Sie können Elemente erstellen, die nur in der Darstellung einer Seite und nicht im DOM selbst erscheinen. Web Components werden einen Weg zur Erstellung von Widgets bereitstellen, die Sie wiederholt in vielen verschiedenen Seiten einer Site hinweg einsetzen können, ohne dass Sie sich dabei um Konflikte mit bestehendem CSS- und JavaScript-Code sorgen müssen – schließlich besitzt das Widget sein eigenes, paralleles DOM.

Die Web-Components-Spezifikation ist derzeit immer noch im Entwurfsstadium. Ich werde die Konzepte deshalb nicht im Detail beleuchten, wohl aber auf Grundlagen eingehen, weil sie so wichtig werden könnten.

11.1.1 Templates

Templates bieten vielleicht den besten Einstieg zum Verständnis von Web Components. Die Idee, bei der Entwicklung wiederverwendbare Codeblöcke (oder *templates*) einzusetzen, ist in der Webentwicklung seit einiger Zeit fest verwurzelt – auch wenn wir bisher nie eine native

Implementation in HTML gesehen haben. Zum Einsatz von Templates waren bisher immer serverseitige Sprachen oder JavaScript (wie etwa die Mustache-Bibliothek aus Kapitel 5) erforderlich.

Stellen Sie sich ein Web-Components-Template als eine Art inaktiven DOM-Bereich vor. Das Wichtige daran ist, dass die Inhalte vom Browser geparst, aber nicht dargestellt werden. Bilder und andere externe Elemente werden also nicht geladen und enthaltene Skripte werden nicht ausgeführt. Im Vergleich zu per CSS ausgeblendeten Elementen, die aber dennoch geladen werden, kann dies einen gewaltigen Leistungsschub ergeben.

template

Ein Template wird mit dem `template`-Element deklariert, und jegliche Unterelemente ergeben den Inhalt des Templates. Der nachfolgende Codeblock zeigt ein Template-Element mit der `id` *#foo*, das zwei Unterelemente besitzt (ein `h2`- und ein `p`-Element). Außerhalb des Templates befindet sich ein `div`-Element mit der `id` *#bar*, das ein `h1`-Element enthält.

```
<template id="foo">
  <h2>Gorilla Beringei</h2>
  <p>A species of the genus Gorilla...</p>
</template>
<div id="bar">
  <h1>Eastern Gorilla</h1>
</div>
```

Würden Sie diese Seite mit den Entwicklerwerkzeugen Ihres Browsers betrachten, könnten Sie das `template`-Element ohne Inhalte sehen, da die Inhalte dieses Elements für das DOM im Grunde unsichtbar sind.

content

Per Skript greifen Sie auf das Template über das `content`-Objekt zu, welches die untergeordneten Elemente des Templates als HTML-Fragment zurückliefert. Im nächsten Codeabschnitt wird das Template zum Beispiel der Variablen *tpl* zugewiesen und das `content`-Objekt in der Konsole ausgegeben:

```
var tpl = document.getElementById('foo');
console.log(tpl.content);
```

Sobald Sie das Fragment haben, können Sie es nach Belieben manipulieren. Der nachfolgende Code legt mittels `cloneNode()` einen Klon des Inhalts an und fügt diesen mittels `appendChild()` in *#bar* ein:

```
var bar = document.getElementById('bar'),
   clone = tpl.content.cloneNode(true);
bar.appendChild(clone);
```

An dieser Stelle würden Sie, wenn Sie das DOM inspizieren, das folgende Markup finden:

```
<template id="foo"></template>
<div id="bar">
  <h1>Eastern Gorilla</h1>
</div>
```

Die Seite würde jedoch dargestellt, als hätten Sie dieses Markup verwendet:

```
<div id="bar">
  <h1>Eastern Gorilla</h1>
  <h2>Gorilla Beringei</h2>
  <p>A species of the genus Gorilla...</p>
</div>
```

Sie können es sich selbst in der Beispieldatei *templates.html* ansehen. Die Ausgabe ist in Abb. 11–1 dargestellt (im Anhang A finden Sie Informationen zur aktuellen Browserunterstützung). Beachten Sie, dass ich in meinen Entwicklerwerkzeugen die Option *Show Shadow DOM* aktivieren musste, damit die Inhalte des `template`-Elements im DOM dargestellt wurden; wäre diese Option ausgeschaltet, würde das Element leer erscheinen.

Abb. 11–1
Der Code-Inspektor zeigt die Inhalte des Template-Elements, das außerhalb des regulären DOM existiert.

Code, der innerhalb des `template`-Elements funktionslos war, tritt in Aktion, sobald dieses in ein anderes DOM-Element eingefügt wird. Dann werden sämtliche externen Ressourcen geladen, Skripte geparst und so weiter.

11.1.2 Decorator

Benutzerdefiniertes Markup

Decorator erweitern den Nutzen der Templates durch die Möglichkeit, benutzerdefiniertes Markup durch CSS hinzuzufügen. Decorator verwenden das `decorator`-Element, dem eine eindeutige `id` zugewiesen werden muss. Innerhalb des `decorator`-Elements finden Sie ein `template`-Element mit etwas benutzerdefiniertem Markup und das `content`-Element. In diesem wird das Element, auf das die Regel angewendet wird, dargestellt. Unklar? Ich habe auch etwas Zeit gebraucht, um das zu verstehen.

Lassen Sie uns das in Häppchen unterteilen. Der nachfolgende Code zeigt ein Beispiel für einen Decorator. Ich habe ihm (zur Abwechslung) die eindeutige `id` *#foo* zugewiesen. Darin befindet sich ein Template mit einem `div`-Tag, in dem sich das `content`- und ein `h2`-Element befinden.

```
<decorator id="foo">
  <template>
    <div>
      <content></content>
      <h2>A great ape!</h2>
    </div>
  </template>
</decorator>
```

Stellen Sie sich nun vor, im Hauptdokument hätte ich ein `h1`-Element mit der `id` *#bar*, so wie im nachfolgenden Code:

```
<h1 id="bar">Gorilla</h1>
```

Ich wende den Decorator mit CSS und der neuen `decorator`-Eigenschaft an, die als Wert eine `url()`-Funktion mit der `id` des Decorators besitzt.

```
h1#bar { decorator: url(#foo); }
```

Anschließend wird das im Template *#foo* enthaltene Markup zum Markup des Elements *#bar* hinzugefügt, wobei *#bar* selbst das `content`-Element von *#foo* ersetzt. Dies geschieht allerdings erst bei der Darstellung, und das DOM wird dadurch nicht verändert. Bei einer Inspektion enthält das DOM nur das Element *#bar,* und doch wird das Element so dargestellt, als würde folgendes Markup verwendet:

```
<div>
  <h1 id="bar">Gorilla</h1>
  <h2>A great ape!</h2>
</div>
```

Templates und Decorator können aber noch mehr – um Ihnen das zu zeigen, muss ich zunächst einen kleinen Exkurs einfügen und auf *Scoped Styles* eingehen.

Scoped Styles

Vererbungs- und Namenskonflikte

Eine der größten Stärken von CSS ist die *Vererbung* – durch die Regelkaskade können Werte auf mehrere unterschiedliche Elemente angewandt werden. Diese Stärke kann allerdings auch zum Nachteil werden, wenn Sie auf großen Sites mit vielen Stylesheets arbeiten. Vererbungs- und Namenskonflikte, die ich bereits zu Beginn dieses Abschnitts erwähnt hatte, sind hier nicht ungewöhnlich.

scoped

Das Attribut *scoped* bietet eine Möglichkeit, diese Konflikte zu vermeiden. Werden im Dokument `style`-Elemente mit dem Attribut `scoped` verwendet, vererben sich alle darin enthaltenen Regeln nur auf die entsprechenden Unterelemente des betreffenden Elements – ihre Reichweite ist also begrenzt, und anderswo im Dokument werden sie nicht angewendet.

Im nachfolgenden Code sehen Sie dies in Aktion: Ein `style`-Tag mit dem Attribut `scope` wird innerhalb eines `div`-Elements angewendet, und die auf das `h1`-Element angewendeten Regeln gelten nur für das `h1`-Tag innerhalb dieses Elements (mit der `id` *#foo*) und nicht für dasjenige außerhalb des `div`-Tags (mit der `id` *#bar*). Durch das Attribut `scoped` gilt die Regel nur für die Unterelemente des `div`-Elements.

```
<div>
  <style scoped>
    h1 {
      background-color: #333;
      color: #FFF;
    }
  </style>
  <h1 id="foo">Scoped</h1>
</div>
<h1 id="bar">Not Scoped</h1>
```

Sehen Sie sich die Beispieldatei *scoped-style.html* an. Das `h1`-Tag mit der `id` *#bar* steht im DOM nach demjenigen mit der `id` *#foo*, Sie würden also erwarten, dass die Regeln im `style`-Element für beide gelten. Tatsächlich bewirkt das Attribut `scoped` aber, dass die Regeln nur innerhalb des übergeordneten `div`-Elements Anwendung finden. Das Ergebnis sehen Sie in Abb. 11–2 und in *scoped-style.html*.

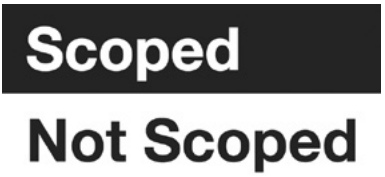

Abb. 11–2
Die Regeln des ersten h1-Elements gelten nicht für das zweite h1-Element, weil die Regeln nur auf das übergeordnete div-Element angewandt werden.

Scoped Styles und Templates

Diese Möglichkeit, den Geltungsbereich von Stilen zu beschränken, eignet sich ideal zur Datenkapselung. Besonders gut lässt sie sich mit Templates und Decorators verbinden. Ein typisches Beispiel wäre etwa der erste Codeblock im Abschnitt »Decorator« auf Seite 237: Hier könnte ich einen Satz Regeln definieren, der nur dann auf das ursprüngliche `h1`-Element angewandt wird, wenn der Decorator mit einem durch das `scoped`-Attribut in seiner Reichweite begrenzte `style`-Tag innerhalb des `template`-Elements angewendet wird:

```
<decorator id="foo">
  <template>
    <div>
      <style scoped>
      h1 { color: red; }
      </style>
      <content></content>
      <h2>A great ape!</h2>
    </div>
  </template>
</decorator>
```

In diesem Fall wird das `h1`-Element nur dann rot eingefärbt, wenn der Decorator angewandt wird. Besser noch: Diese Farbe wird wegen ihres beschränkten Geltungsbereichs auf keines der nachfolgenden `h1`-Elemente im Dokument angewandt – ein perfektes Beispiel für Datenkapselung.

11.1.3 Custom Elements

Decorators sind nützlich, um ein Element mit etwas zusätzlichem Markup anders zu präsentieren – wenn Sie aber tiefer gehende Änderungen anstreben, verwenden Sie ein *Custom Element*. Der wichtigste Unterschied zwischen Custom Elements und Decorators ist, dass Letztere vorübergehender Natur sind; sie lassen sich durch Änderung eines Attributs oder Selektors anwenden oder entfernen. Custom Elements hingegen sind beständig; sie werden beim Parsen des DOM angewandt und lassen sich nur durch Skripte ändern oder entfernen.

Erweiterte Templates, die ein Standardelement verbessern oder ersetzen

Ein Custom Element ist wie ein erweitertes Template, das ein Standardelement verbessert oder ersetzt. Ein solches benutzerdefiniertes HTML-Element erstellen Sie mit dem `element`-Element (dieser Absatz stellt einen Rekordversuch bei der Verwendung des Worts »Element« dar), welches über einige neue Attribute verfügt, die ich in Kürze erläu-

tern werde. Innerhalb dieses Elements können Sie ein `template`-Element mit neuem Markup hinzufügen, ebenso wie mit dem Attribut `scoped` versehene Styles oder gar ein Skript.

Klingt das etwas verwirrend? Der nachfolgende Codeschnipsel zeigt ein einfaches Beispiel: ein `element` mit einem `template` darin, welches wiederum ein `div`-Tag enthält, in dem sich seinerseits das von mit auf Seite 237 in »Decorator« vorgestellte `content`-Element befindet. Das `element` hat zwei Attribute: `extends`, das als Wert den Namen des zu erweiternden Elements übernimmt (in diesem Fall handelt es sich um ein `button`-Element), und `name`, ein benutzerdefinierter und eindeutiger ID-Wert (der mit *x-* beginnen muss, um Konflikte zu bestehenden Elementen zu vermeiden).

```
<element extends="button" name="x-foobutton">
  <template>
    <div id="foo">
      <content></content>
    </div>
  </template>
</element>
```

Nach der Definition des Custom Element können Sie es mit dem `is`-Attribut auf ein bestehendes Element anwenden. Das `is`-Attribut wird auf das zu erweiternde Element angewandt und es akzeptiert als Wert den eindeutigen Bezeichner aus dem für das Custom Element definierten `name`-Attribut (*x-foobutton*). Tatsächlich ist es nicht so kompliziert, wie es vielleicht klingen mag:

```
<button is="x-foobutton">Go</button>
```

Das Ergebnis ist dasselbe wie bei einem Decorator: Das Markup des Custom Element ergänzt das Markup des Elements, auf das es angewandt wurde, allerdings nur in der fertigen Darstellung. Beim Blick ins DOM zeigt sich zwar nur das `button`-Element, es wird jedoch so dargestellt:

```
<div id="foo">
  <button>Go</button>
</div>
```

Das Beispiel ist simpel; es verdeutlicht jedoch, wie Sie mit den Erweiterungsmöglichkeiten dieser Technik auf einfache Weise komplett maßgeschneiderte Widgets programmieren könnten, die sich über viele Dokumente hinweg wiederverwenden ließen. Im Ergebnis ließen sich viele unserer sperrigen Widgets (wie Karusselle, Akkordeons und Datumswähler) auf bestehende Elemente anwenden, ohne das DOM mit unnö-

tigem Code zu füllen – mit den zusätzlichen Vorteilen der Datenkapselung zur Vermeidung von Konflikten.

Ich habe bereits erwähnt, dass der Hauptunterschied zwischen einem Custom Element und einem Decorator in der Beständigkeit des Markups liegt. Ein Vorteil davon ist, dass Skripte in ein Custom Element gepackt werden können und dann immer präsent sind (darauf können Sie sich bei den kurzlebigeren Decorators nicht verlassen). Das alles bedeutet, dass Sie sogar für jedes Custom Element eine imperative API definieren könnten, um damit die Interaktivität auf eine völlig neue Ebene zu heben.

11.1.4 Das Shadow DOM

Der letzte Bestandteil der Web-Components-Spezifikation ist das *Shadow DOM*. Das ist nicht nur ein cooler Name für einen Superschurken, sondern auch eine Möglichkeit, mittels Skript die im bereits beschriebenen parallelen DOM befindlichen Elemente zu erstellen und darauf zuzugreifen. Decorator verwenden CSS zur Manipulation von Elementen, Custom Elements greifen dafür auf HTML zurück und das Shadow DOM nutzt zu diesem Zweck Skripte.

Eine neue, abgekapselte Knotenstruktur

Das Shadow DOM beschreibt die Fähigkeit eines Browsers, eine neue, völlig abgekapselte Knotenstruktur im bestehenden DOM zu erzeugen. Dazu erzeugt der Browser eine *Shadow Root* in einem Element. Sie kann wie eine normale Knotenstruktur traversiert und manipuliert werden. (Ein Shadow Tree erscheint nicht im DOM, wird aber dargestellt.)

Jetzt ein Beispiel. Der nachfolgende Codeabschnitt enthält einfaches Markup: ein `div`-Tag namens *#foo*, in dem ein einzelnes `h1`-Element enthalten ist. Dies ist der Basiscode im DOM, in den ich nun eine Shadow Root einfügen werde.

```
<div id="foo">
  <h1>Hello, world!</h1>
</div>
```

Jetzt füge ich eine neue Shadow Root in das `div`-Element ein und hänge dann ein neues Element an die neue Wurzel an. Nachfolgend erkläre ich diesen Code Schritt für Schritt.

```
var foo = document.getElementById('foo'),
❶   newRoot = foo.createShadowRoot(),
❷   newH2 = document.createElement('h2');
 newH2.textContent = 'Hello, shadow world!';
❸ newRoot.appendChild(newH2);
```

Zunächst wurde eine neue Shadow Root in *#foo* erstellt ❶. Dazu dient die Methode `createShadowRoot()`. In den beiden nachfolgenden Zeilen ❷ lege ich ein neues `h2`-Element mit dem Textinhalt *Hello, shadow world!* an. Und schließlich ❸ hänge ich das neue `h2`-Element der neuen Shadow Root an.

Wenn dieser Code ausgeführt wird, sehen die Anwender ein `h2`-Element mit dem Text *Hello, shadow world!*. Würden sie jedoch das DOM betrachten, könnten sie den Originalinhalt *Hello, world!* sehen. Das `h1`-Element wurde komplett vom neuen Shadow-Baum ersetzt. Auf das DOM hat dies keinen Einfluss.

Abb. 11–3 zeigt, wie dies in den Chrome-Entwicklertools dargestellt wird. Die Inhalte der Shadow Root werden in einer neuen Knotenstruktur unter dem Bezeichner *#shadow-root* angezeigt.

Abb. 11–3
Die Shadow-Root ist im DOM-Baum deutlich gekennzeichnet.

Wenn Sie den Inhalt des Elements, in dem Sie eine neue Wurzel angelegt haben, nicht ersetzen möchten, können Sie wieder auf das `content`-Element zurückgreifen (auch das habe ich bereits im Abschnitt »Decorator« auf Seite 237 vorgestellt), um die Originalelemente einzufügen. Ich veranschauliche das im folgenden Code, mit dem ich das `content`-Element erstelle und es dann an die neue Shadow Root anhänge. Im Ergebnis sieht der Anwender zuerst das neue Schatten-`h2` und danach folgt das ursprüngliche `h1`-Element – obwohl im DOM nur das `h1`-Element zu sehen ist.

```
var content = document.createElement(content);
newRoot.appendChild(content);
```

Sie können auch Templates zusammen mit Shadow Trees verwenden. Hier sehen Sie beispielsweise, wie Sie ein HTML-Fragment aus dem template *#foo* an das Shadow DOM anhängen.

```
var foo = document.getElementById('foo');
newRoot.appendChild(foo.content);
```

Das Shadow DOM geht sogar noch weiter als dieses einfache Beispiel, und es ist ein sehr flexibles und leistungsfähiges Werkzeug. Ich kann ihm hier jetzt nicht mehr Zeit widmen, aber im Abschnitt »Literaturempfehlungen« auf Seite 255 finden Sie einige Links zu detaillierteren Artikeln.

11.1.5 Die Puzzleteile zusammenfügen

Ich habe nur an der Oberfläche der Web-Components-Spezifikation gekratzt, aber ich hoffe, Sie konnten durch diese Einblicke schon Feuer fangen. Mit Web Components stehen vollständig wiederverwendbare Codebestandteile zur Verbesserung bestehender Elemente vor der Tür. Diese Komponenten werden vollständig vom Rest des Dokuments abgekapselt sein; der Browser stellt sie dar, über das DOM sind sie jedoch nicht ansprechbar. Ein paralleles Schatten-DOM erlaubt jedoch vollständigen Zugriff auf die Elemente innerhalb jeder einzelnen Komponente und entsprechende Veränderungsmöglichkeiten.

Wenn die Web-Components-Spezifikation implementiert ist, wird sie unsere Herangehensweise bei der Entwicklung von Anwendungen und Websites vollkommen umkrempeln. Und wenn Sie das nicht spannend finden, sind Sie vielleicht in der falschen Branche unterwegs!

11.2 Die Zukunft von CSS

CSS3 hat das Web in verschiedener Hinsicht revolutioniert – etwa mit der Einführung von dynamischen Elementen wie Übergängen und Animationen und durch echte Lösungen von Layoutproblemen im Web. Diese Funktionen sind aber nur die Spitze des Eisbergs zukünftiger Möglichkeiten. Viele große Technologieunternehmen sehen ihre Zukunft im Web, und ihr Engagement bei der Gestaltung zukünftiger Webstandards bringt einen enormen Innovationsschub mit sich.

Adobes umfassende Wendung in Richtung offener Standards (eine angenehme Überraschung!) macht das Unternehmen beispielsweise zu einem bedeutenden Mitspieler bei der Browserentwicklung, und seine Erfahrung im Grafik- und Publishing-Bereich fließt mit in CSS ein. Als erste Früchte dieser Arbeit ermöglichen CSS-*Regions* und -*Exclusions* zukünftig eine grundlegend andere Herangehensweise an das Seitenlayout, und dynamische Websites können es endlich mit dem Desktop Publishing aufnehmen.

Die größte Triebkraft bei der Entwicklung von CSS hat jedoch die Webentwickler-Gemeinde. Von Entwicklern erstellte JavaScript-Bibliotheken wie jQuery und Modernizr haben einen direkten Einfluss auf die Sprache, wie Sie bereits am Beispiel von `querySelector()` und `querySelectorAll()` (in Kapitel 5) sehen konnten. Und dieser Einfluss wird mit der Einführung der *Feature Queries* weiterhin deutlich zu spüren sein.

Zudem gewöhnen sich Frontend-Entwickler durch die wachsende Beliebtheit von CSS-Präprozessoren wie Sass und LESS zunehmend an den Einsatz von Programmierprinzipien wie Variablen und Funktionen. Der Bedarf, diese in der Sprache zu verankern, manifestiert sich durch *kaskadierende Variablen*.

11.2.1 Regions

Flexibler Spaltensatz

In Kapitel 4 habe ich den CSS-Spaltensatz angesprochen, mit dem sich Inline-Inhalte in Spalten unterteilen lassen, sodass sie von der ersten in die zweite und in die dritte Spalte fließen usw. Stellen Sie sich vor, die Spalten würden nicht direkt nebeneinander liegen – die erste befindet sich links auf der Seite, die zweite liegt rechts und die dritte am unteren Seitenrand –, aber der Inhalt durchfließt trotzdem nacheinander die Spalten. Darum geht es im Kern beim neuen CSS-Konzept *Regions*.

Regions funktionieren so: Ein Element wird zur Quelle erklärt und der Inhalt dieser Quelle fließt in ein weiteres Element oder in eine Reihe weiterer Elemente, die sogenannte *Region Chain*. In der Praxis bedeutet dies, dass Ihr Inhalt nacheinander mehrere Elemente durchfließen kann, und das unabhängig von deren Reihenfolge im DOM.

Hier ist ein kleines Beispiel, das mit drei `div`-Elementen anfängt: einmal *#foo* und zweimal *.bar*. Das erste, *#foo*, erhält einen Inhalt, die beiden anderen bleiben leer:

```
<div id="foo">
  <p>...</p>
</div>
<div class="bar"></div>
<div class="bar"></div>
```

Im nächsten Schritt holen Sie den Inhalt aus *#foo* und kopieren ihn in eine benannte Textkette, einen sogenannten *Named Flow*, als würden Sie ihn etwa in einer Variablen ablegen (oder in der Zwischenablage Ihres Computers). In CSS-Regions erzeugen Sie diesen Named Flow durch Deklaration der `flow-into`-Eigenschaft für das Quellelement (*#foo*). Als Wert für die Variable können Sie einen beliebigen eindeutigen Namen wählen, ich nenne meine also *myFlow*. Nachdem ich mein Flow (oder meine Zwischenablage, um bei der Analogie zu bleiben) benannt habe,

kann ich es in andere Elemente hineinfließen lassen, die unter dem Titelnamen »*Regions*“ geführt werden:

```
#foo { flow-into: myFlow; }
```

WARNUNG
Im Internet Explorer 10 muss das Quellelement ein `iframe` sein, und der Inhalt im `body` des verlinkten Dokuments wird zum Flow-Inhalt.

Wenn diese Eigenschaft angewendet wird, werden das Quelleelement und seine untergeordneten Elemente nicht mehr auf dem Bildschirm ausgegeben, obwohl sie im DOM nach wie vor sichtbar und zugreifbar sind.

Als Nächstes muss ich eine Region Chain definieren, also die Bereiche, die der Inhalt durchfließen soll. Für jedes Element in der Region Chain sollte die `flow-from`-Eigenschaft deklariert werden. Als Wert dient dabei zuvor definierte Named Flow. Der nachfolgende Code zeigt, wie der Inhalt des Flows *myFlow* in alle mit *.bar* benannten Regionen hineinfließt.

```
.bar { flow-from: myFlow; }
```

Der Inhalt in *myFlow* fließt in DOM-Reihenfolge durch die Region Chain. Zunächst fließt er in die erste Instanz von *.bar*, sämtlicher Übersatz fließt in die zweite Instanz von *.bar* und so weiter, bis der Inhalt erschöpft ist. Probieren Sie es in der Datei *regions.html* aus (siehe auch Abb. 11–4).

Wie ich bereits zu Anfang dieses Abschnitts erwähnt habe, funktionieren CSS-Regions wie ein mehrspaltiges Layout, wobei die Spalten aber nicht unmittelbar nebeneinander liegen müssen. Dieses neue Element bietet faszinierende Möglichkeiten zur Erstellung dynamischer und interessanter Seitenlayouts, die von der jahrelangen Erfahrung im Bereich gedruckter Medien inspiriert sind.

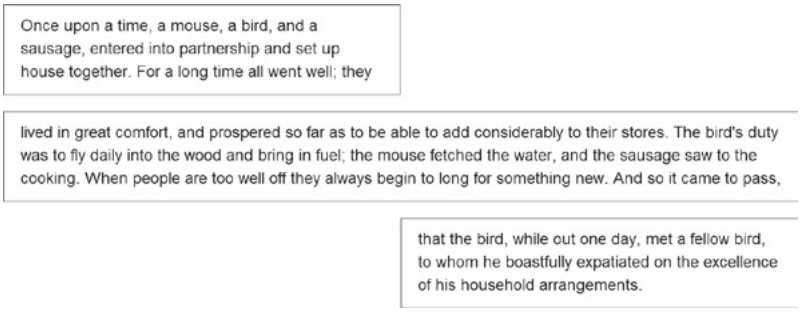

Abb. 11–4
Mit CSS-Regions können Sie Inhalt durch mehrere Elemente fließen lassen, die nicht direkt beisammen liegen müssen.

11.2.2 Exclusions

Textfluss

CSS Exclusions können Sie sich als eine Art positionierter Float-Elemente vorstellen – ein früheres Konzept hat sie tatsächlich als genau das beschrieben. In CSS2.1 finden Sie Float-Elemente nur auf der linken Seite und der übrige Inhalt umfließt sie dabei rechts oder umgekehrt. Mit CSS Exclusions soll es jedoch möglich werden, ein Element von Inhalt umfließen zu lassen, egal an welcher Stelle der Seite es sich befindet.

Sehen Sie sich zur Veranschaulichung folgendes Markup an, das das Container-Element *#foo*, etwas Textinhalt in einem `p`-Tag und das untergeordnete Element *#bar* enthält.

```
<div id="foo">
  <p>...</p>
  <div id="bar"></div>
</div>
```

Ich möchte *#bar* absolut über dem Inhalt in *#foo* platzieren. Hierzu benötige ich etwa diese Style-Regeln:

```
#foo { position: relative; }
#bar {
  left: 20px;
  position: absolute;
}
```

Hier liegt *#bar* auf einer über den Textinhalt gelegten Ebene. Wie in Abb. 11–5 zu sehen ist, verdeckt es alles dahinter Liegende. Ich möchte aber, dass *#bar* Teil derselben Ebene wird und dass es vom Text umflossen wird. *#bar* soll also ein *Exclusion Element* werden.

Genau das kann ich durch Anwendung der `wrap-flow`-Eigenschaft erreichen. Jeder untergeordnete Inhalt umfließt dadurch das neue Exclusion-Element entsprechend des Werts der Eigenschaft.

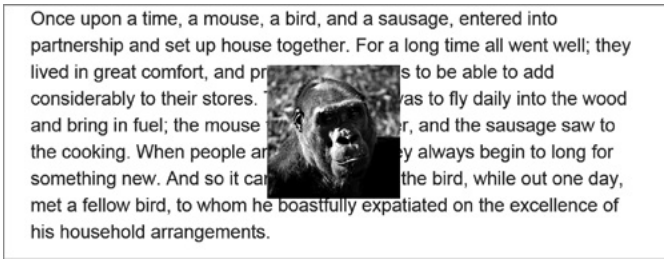

Abb. 11–5
Ein absolut platziertes Element beeinflusst den Textfluss des darunter liegenden Inhalts nicht.[1]

1 Dieses Gorillafoto stammt von Chris Willis und ist unter http://www.fotopedia.com/items/flickr-3059127796/ zu sehen. Es steht unter einer Creative-Commons-Lizenz.

Das nachfolgende Beispiel verwendet den Wert `both`, um das Exclusion-Element beidseitig von Text umfließen zu lassen:

```
#bar { wrap-flow: both; }
```

Den Unterschied erkennen Sie in Abb. 11–6. Die Elemente sind genau wie vorher platziert; weil *#bar* nun aber zum Exclusion-Element deklariert wurde, wird es vom Inhalt in *#foo* beidseitig umflossen.

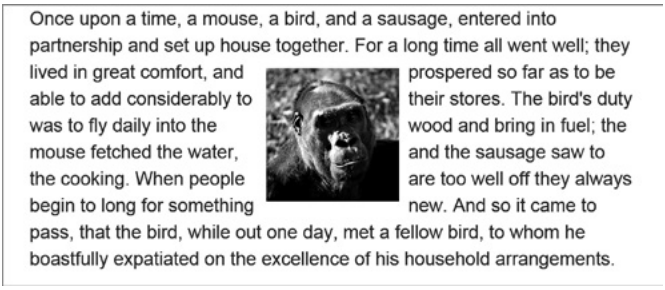

Abb. 11–6
Wenn Sie die wrap-flow-Eigenschaft anwenden, wird das Element im Umflussverhalten des Dokuments berücksichtigt und der Inhalt umfließt es auf beiden Seiten.

Weitere mögliche Werte für `wrap-flow` sind etwa:

- `start`, um den Inhalt (in von links nach rechts geschriebenen Sprachen) links vom Exclusion-Element umfließen zu lassen, aber nicht rechts davon
- `end` für das Gegenteil
- `maximum` und `minimum`, um den Inhalt jeweils nur auf der Seite mit dem meisten (oder wenigsten) Platz zwischen dem Inhalt und seinem Container-Element umfließen zu lassen
- `clear`, um Inhalt nur über und unter dem Exclusion-Element umfließen zu lassen

Abb. 11–7 zeigt ein paar unterschiedliche Werte im Einsatz. Das erste Beispiel hat den Wert `start`, also umfließt der Inhalt das Exclusion-Element links. Das nächste verwendet den Wert `end`, der Inhalt umfließt das Element also auf der entgegengesetzten Seite. Und im letzten Beispiel sorgt der Wert `clear` dafür, dass das Element seitlich überhaupt nicht umflossen wird.

Damit Sie den Umfluss von Inline-Elementen um das Exclusion-Element besser steuern können, gibt es die `wrap-through`-Eigenschaft. Diese akzeptiert zwei verschiedene Werte: `flow` und `none`. Der zuerst genannte ist der Standardwert; er sorgt dafür, dass Inline-Inhalte das Exclusion-Element umfließen. Der zuletzt genannte erlaubt dies nicht. Das ist hilfreich, wenn Sie den Inhaltsumfluss auf Einzelelementbasis steuern möchten.

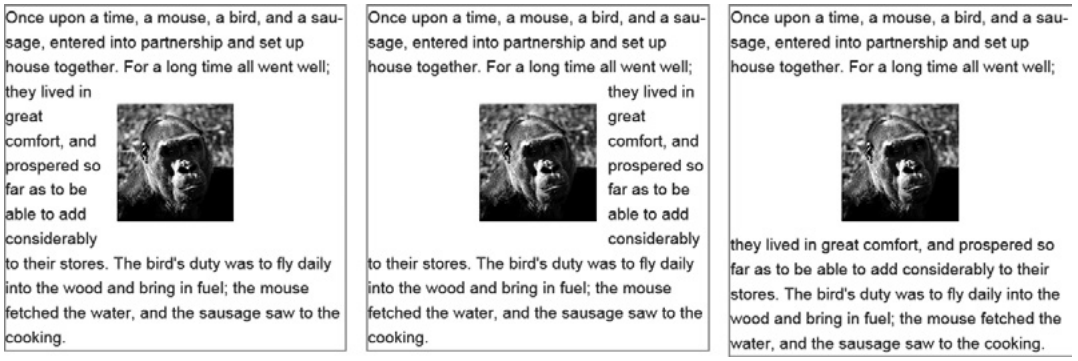

Abb. 11–7
Je nachdem, welche Werte auf wrap-flow angewendet werden, umfließt der Inhalt das Exclusion-Element auf verschiedenen Seiten.

Exclusions und Grids

Für mich ist das Spannendste an CSS Exclusions ihr Zusammenspiel mit dem Modul Grid Layout, das ich in Kapitel 4 vorgestellt habe. Jedes Grid-Element lässt sich in ein Exclusion-Element verwandeln, was die Möglichkeiten von Rasterlayouts enorm erweitert. Als einfaches Beispiel gehen Sie von einem Raster aus drei Spalten und zwei Zeilen aus:

```
E {
  display: grid;
  grid-columns: 1fr 1fr 1fr;
  grid-rows: 100px 1fr;
}
```

Auf diesem Raster platziere ich zwei Elemente mit überlappenden Zellen (sie überlappen in Reihe zwei, Spalte zwei):

```
F, G { grid-column-span: 2; }
F {
  grid-column: 2;
  grid-row-span: 2;
}
G { grid-row: 2; }
```

Nach den üblichen Platzierungsregeln würde Element `G` über Element `F` gestapelt, da es im DOM erst später auftaucht. Indem wir aber Element `G` zum Exclusion-Element erklären, umfließt es der Inline-Inhalt von Element `F`:

```
G {
  grid-row: 2;
  wrap-flow: both;
}
```

Wie Sie in der Beispieldatei *grid-exclusion.html* (und in Abb. 11–8) sehen, nimmt `F` nun beim Umfließen von Element `G` etwa die Form eines

umgekehrten L ein. Dieses Layoutmuster findet sich recht häufig in gedruckten Medien, und daher ist es wirklich eine Steigerung, dass Sie es nun auch im Webdesign einsetzen können. Mit Gestaltungsparadigmen, die beim Buch- und Magazinsatz schon seit Hunderten von Jahren selbstverständlich angewendet werden, läuten CSS Exclusions ein komplett neues Zeitalter für das Layout von Webseiten ein.

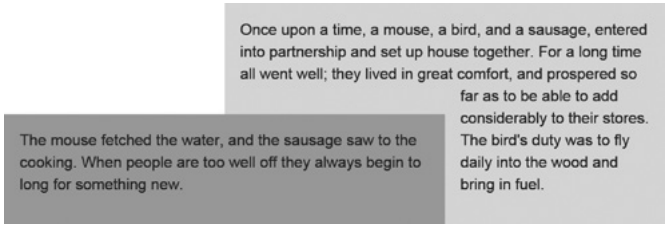

Abb. 11–8
CSS Exclusions arbeiten nahtlos mit CSS-Rastern zusammen. So werden Layoutmuster ermöglicht, die zuvor nicht umsetzbar waren.

Shaped Exclusions

CSS Shapes

Die in den Beispielen dieses Abschnitts verwendeten CSS Exclusions verwendeten bisher normale Block-Elemente; sie erscheinen kastenförmig. In Zukunft sollen sie aber nicht auf rechteckige Exclusions beschränkt bleiben, denn zwei neue Eigenschaften sollen Ihnen das Zeichnen geometrischer Exclusion-Formen ermöglichen. Diese Funktionalität wird inzwischen in einer separaten Spezifikation behandelt, nämlich als »CSS Shapes« (*http://www.w3.org/TR/css-shapes-1/*).

Die Eigenschaften `shape-inside` und `shape-outside` werden als Wert entweder eine geometrische Grundform wie einen Kreis oder eine Ellipse akzeptieren oder auch eine Reihe von Koordinaten, aus denen eine völlig freie Polygonform entsteht. Inhalte könnten die Form dann entweder von außen oder von innen umfließen (oder beides). Damit bieten sich Möglichkeiten für differenzierte Layouts, wie sie im Druck schon lange üblich sind und die sich nun mit der Dynamisierung der Webinhalte immer weiter verbessern.

11.2.3 Layouts: Noch weiter in die Zukunft geblickt

Während ich dies schreibe, befindet sich eine Reihe neuer Regeln und Eigenschaften in verschiedenen Implementierungsstadien – von kaum vorhanden bis hin zu reinen Vorschlägen. Ich hoffe, dass sie alle übernommen und umgesetzt werden, weil sie Lösungen für unterschiedliche Probleme bieten. Die CSS-Spezifikation befindet sich ständig im Fluss, und nichts kann als selbstverständlich angesehen werden; diese Regeln und Eigenschaften könnten umgesetzt werden, sie könnten teilweise

und mit abweichender Syntax oder auch überhaupt nicht implementiert werden.

Es lohnt sich meiner Ansicht nach trotzdem, einen Blick darauf zu werfen. Dafür sprechen zwei Gründe: Zunächst einmal können Sie daran erkennen, wie über Lösungen zu den Problemen des Weblayouts nachgedacht wird – und außerdem werden Sie sie im Fall ihrer Implementierung vielleicht verwenden müssen.

Box Alignment

Mit dem Box-Alignment-Modul wird eine über viele Module hinweg einheitliche Syntax zur Ausrichtung von Elementen innerhalb ihres übergeordneten Elements angestrebt. Als Inspiration für Box Alignment dient die Flexbox-Syntax: `justify`-Eigenschaften werden dabei für die Ausrichtung an der Inline- und Hauptachse verwendet (inline/main axis), `align`-Eigenschaften für die Stapelausrichtung/querende Achse (stacking/cross-axis). Um beispielsweise ein Element entlang seiner Hauptachse auszurichten, würden Sie die `justify-self`-Eigenschaft verwenden; zur Ausrichtung der untergeordneten Elemente eines Elements entlang der Querachse käme dann `align-content` zum Einsatz.

Line Grid

Neben dem wohlbekannten Raster aus Spalten und Zeilen verwenden Typografen häufig auch einen sogenannten *vertikalen Rhythmus*. Das ist ein zweites Raster, das sich aus den Textzeilen und Überschriften auf einer Seite ergibt. Wenn Sie einen vertikalen Rhythmus verwenden, versuchen Sie die senkrechten Abmessungen und Ausrichtungen von Objekten im Sinne einer besseren Lesbarkeit harmonisch zum Text zu gestalten.

Das Modul Line Grid erzeugt auf der Grundlage von `font-size` und `line-height` von Textelementen einen vertikalen Rhythmus. An diesem virtuellen Linienraster können Sie dann Ihre Objekte besser ausrichten. Block-Elemente lassen sich an festen Gitterpunkten einrasten, und somit lassen sich die von der Browser-Engine vorgegebenen Standardpositionen übergehen.

Paged Media

Scrolling ist heute de facto der Standard im Umgang mit umfangreichen Bildschirminhalten. Mit Geräten wie etwa TV-Fernbedienungen und E-Book-Readern mit E-Paper-Displays gestaltet sich das aber nicht immer

einfach. Ein besserer Ansatz für solche Geräte könnte stattdessen eine Einzelseitendarstellung sein.

Mit den im Modul Paged Media vorgeschlagenen Funktionen können Sie das leicht umsetzen. Das Modul führt die Eigenschaft `overflow-style` ein. Ein Wert von `paged-x` oder `paged-y` legt automatisch eine horizontale (oder vertikale) Seitenfolge an, während `-controls` (wie etwa `paged-x-controls`) vom Browser erstellte Bildschirmbedienelemente beifügt, falls die entsprechende Nutzerschnittstelle sie benötigt.

Ein anderer Vorschlag sind Seitenvorlagen, die dieses Konzept noch erweitern. Auf allen Seiten fest verankerte Inhaltsbereiche ermöglichen eine einheitliche Leseerfahrung und reichhaltige interaktive Layouts im Magazinstil.

11.2.4 Feature Queries

In Kapitel 5 habe ich über die JavaScript-Bibliothek Modernizr gesprochen. Mit ihr lassen sich bestimmte Fähigkeiten im Browser des Anwenders aufspüren. Ich erwähnte auch kurz die native Adaption in CSS durch die `@supports`-Regel. Auf diese möchte ich nun etwas näher eingehen, denn sie ist extrem nützlich und findet schnellen Einzug in die Browser.

@supports

Die `@supports`-Regel verhält sich wie eine Medienabfrage: Sie erstellen eine logische Abfrage, und falls diese »true« zurückmeldet, werden die Regeln innerhalb der nachfolgenden Klammern angewendet. Anstelle von Medienfunktionen werden hier allerdings CSS-Eigenschaft-Wert-Paare, sogenannte *Feature Queries*, abgefragt. Um beispielsweise zu prüfen, ob der Browser eines Anwenders die `column-count`-Eigenschaft unterstützt und Sie dementsprechende Stile anbieten können, ließe sich eine Abfrage wie die folgende konstruieren:

```
@supports (column-count: 1) { ... }
```

Wie bei den Media Queries können Sie mithilfe logischer Operatoren auch komplexere Abfragen erstellen. Die nachfolgende verwendet den `and`-Operator und stellt damit Browsern mit Unterstützung für die beiden Eigenschaften `column-count` und `box-sizing` Stile zur Verfügung:

```
@supports (column-count: 1) and (box-sizing: border-box) { ... }
```

Mit dem `or`-Operator können Sie auch Abfragen erstellen, die bestimmte Funktionen aufdecken. Das ist extrem nützlich, wenn Sie herstellerspezifische Eigenschaften mit entsprechendem Präfix berücksichtigen. Im nächsten Beispiel werden sowohl die `hyphens`- als auch die `-moz-hy`-

phens-Eigenschaften abgeprüft. Sobald eine davon unterstützt wird, kommt die Regel zur Anwendung:

```
@supports (-moz-hyphens: auto) or (hyphens: auto) { ... }
```

Mit dem not-Operator können Sie Stile für Browser bereitstellen, die eine bestimmte Eigenschaft nicht unterstützen. (Im Gegensatz zu den anderen Operatoren muss dieser in runden Klammern stehen.)

```
@supports (not (-webkit-hyphens: auto)) { ... }
```

Feature Queries bieten eine API, die ebenso leicht wie die @supports-Regel einzusetzen ist. Sie können beispielsweise mithilfe der CSS.supports()-Methode auf eine Einzelfunktion testen, indem Sie ein Eigenschaft-Wert-Paar als zwei Einzelargumente übergeben. Hier wird der flex-Wert für die display-Eigenschaft abgeprüft:

```
var supports = CSS.supports('display','flex');
```

Und Sie können auch komplette Abfragen als Einzelargument übergeben, wenn Sie diese als Zeichenkette in Hochkommas setzen:

```
var supports = CSS.supports('(column-count: 1) and (display: flex)');
```

Das Modernizr-Projekt hat bereits begonnen, dies in seine Bibliothek zu implementieren. Falls eine native Unterstützung für CSS.supports() gegeben ist, greift das Skript darauf zu, falls nicht, verwendet es die eingebauten Testroutinen von Modernizr.

11.2.5 Cascading Variables

Der Nutzen von Variablen hat sich über die Jahre hinweg in nahezu jeder Programmiersprache erwiesen, trotz regelmäßiger Bitten aus der Community wurden sie jedoch niemals in CSS implementiert. Mit rasch zunehmender Beliebtheit der CSS-Präprozessoren lernt aber eine ganze Coder-Generation Variablen in ihren Stylesheets zu lieben, und die Rufe nach einer nativen Implementierung in die Sprache lassen sich nicht länger ignorieren.

Im aktuellen Vorschlag haben CSS-Variablen eine beschränkte Reichweite. Eine echte Variable lässt jede Art von Wert zu und kann überall im Code eingesetzt werden – etwa um einer Variablen einen Selektor zuzuweisen. Den vorgeschlagenen CSS-Variablen können nur gültige CSS-Werte zugewiesen werden, und sie lassen sich nur als Wert einer Eigenschaft einsetzen. Zur Abgrenzung werden sie daher als *Cascading Variables* bezeichnet.

Custom Property

Jede Cascading Variable wird mit einer *Custom Property* deklariert. Das ist ein mit `var-` beginnender, benutzerdefinierter Eigenschaftsname, dem ein Wert zugewiesen wird. Hier wird der benutzerdefinierten Eigenschaft *var-foo* der Farbwert *#F00* zugewiesen:

```
:root { var-foo: #F00; }
```

Beachten Sie, dass ich zur Deklaration dieser Eigenschaft den `:root`-Selektor verwendet habe. (Ich erkläre gleich noch, warum.)

Um den Wert der benutzerdefinierten Eigenschaft zu verwenden, rufen Sie ihn mit der `var()`-Funktion auf. Der benutzerdefinierte Name (der Teil nach `var-`) steht dabei in runden Klammern. Der Wert der benutzerdefinierten Eigenschaft wird zugleich der aufrufenden Eigenschaft als Wert übermittelt. Im nachfolgenden Listing wird im `h1`-Element zweimal die *var-foo*-Eigenschaft mit der `var(foo)`-Funktion aufgerufen: einmal für die `border-bottom`-Eigenschaft und einmal für `color`. Auf jede Eigenschaft wird dementsprechend der Farbwert *#F00* angewendet.

```
h1 {
  border-bottom: 1px solid var(foo);
  color: var(foo);
}
```

Cascading Variables haben eine begrenzte *Reichweite*, sie gelten also nur für das Element, für das sie deklariert wurden, sowie für mögliche Unterelemente. Ich habe im Beispiel für diesen Absatz den `:root`-Selektor zur Deklaration einer benutzerdefinierten Eigenschaft verwendet – die Variable ist daher *global gültig*: Sie kann auf jedes Seitenelement angewendet werden. Hätte ich einen anderen Selektor verwendet, wäre der Wert der in der benutzerdefinierten Eigenschaft deklarierten Variablen nur in den Unterelementen der entsprechenden Elemente bzw. des entsprechenden Elements gültig.

Im nachfolgenden Codebeispiel wird etwa die benutzerdefinierte Eigenschaft *var-foo* mit dem Wert *#F00* für das `:root`-Element deklariert, doch ich schreibe darunter noch einen anderen Wert und Selektor. In diesem Fall wäre der Variablenwert eigentlich für das ganze Dokument *#F00*, durch die zweite Zeile ergibt sich nun aber für das *.bar*-Element und seine Unterelemente der Variablenwert *#00F*.

```
:root { var-foo: #F00; }
.bar { var-foo: #00F; }
```

Hinweis
Auf längere Sicht werden auch die beliebtesten Präprozessor-`Mixins` in CSS implementiert werden. Ein Mixin gleicht einer erweiterten Variablen, mit deren Hilfe sich Codeabschnitte über mehrere Selektoren hinweg wiederverwenden lassen. Es wurde sogar schon über die Implementierung vollständiger Variablen gesprochen, die dann Eigenschaftsnamen und Selektoren ersetzen könnten.

11.3 Zusammenfassung

In diesem Kapitel haben wir uns einige der etwas experimentelleren Funktionen der Webplattform angesehen. Diese befinden sich noch in der Testphase und können sich daher noch verändern. Sie sind aber so leistungsfähig und potenziell wichtig für die Zukunft der Plattform, dass ich das Buch nicht zu Ende schreiben konnte, ohne sie zu erwähnen.

Als Erstes waren das die Web Components, die größte Veränderung an HTML seit seiner Erfindung. Web Components bestehen aus einer Reihe von Funktionen. Sie erzeugen ein paralleles DOM zur mehrfachen Nutzung von Codeabschnitten, mit deren Hilfe die Standardelemente in HTML erweitert und verbessert werden. Dies geschieht bei vollständiger Datenkapselung, um Konflikte mit anderen CSS-Regeln und JavaScript-Funktionen auszuschließen.

Als Nächstes befassten wir uns mit der Zukunft von CSS, einer Sprache, die dank der Einflussnahme großer Technologiefirmen ebenfalls großen Veränderungen unterworfen ist. CSS Regions und Exclusions werden möglicherweise die benötigten Werkzeuge für dynamische, individuelle Layouts bereitstellen, die allen Umsetzungsmöglichkeiten in Printmedien das Wasser reichen (oder sie sogar übertreffen?) könnten.

Schließlich behandelte ich neue CSS-Funktionen, die auf Grundlage von Innovationen aus der Webentwickler-Community entstehen. Dazu gehören Feature Queries, die CSS eine native Feature-Detektion im Stil von Modernizr bescheren, und Cascading Variables, mit denen die besten Präprozessor-Funktionen allmählich Einzug in die Sprache selbst halten.

11.4 Literaturempfehlungen

Web Components sind in diesem Moment noch ziemlich neu, es gibt dazu also noch nicht so viele Ressourcen. Als Erstes sollten Sie sich dazu die *entwicklerfreundliche Einführung der Verfasser der Spezifikation* ansehen und danach dann vielleicht die *Präsentation von Eric Bidelman*. Beide Quellen vermitteln Ihnen gut die Grundkonzepte. Zu finden sind sie unter http://dvcs.w3.org/hg/webcomponents/raw-file/tip/explainer/index.html und http://html5-demos.appspot.com/static/webcomponents/index.html (eventuell benötigen Sie hier Google Chrome für die korrekte Anzeige).

Das Shadow DOM ist bisher der am besten implementierte Bestandteil der Web Components und verfügt als solcher über eine umfangreichere Online-Dokumentation. Sowohl *Sitepoint* als auch *HTML5 Rocks* liefern gut verständliche Erklärungen zum Thema, diese finden Sie unter http://www.sitepoint.com/the-basics-of-the-shadow-dom/ und http://www..html5rocks.com/en/tutorials/webcomponents/shadowdom/.

Falls Ihr Browser keine Custom Elements unterstützt, sehen Sie sich *X-Tags* an. Diese von Mozilla entwickelte, experimentelle Bibliothek bildet das Verhalten von Custom Elements nach und bietet dabei ein großes Register vorgefertigter Komponenten: http://x-tags.org/.

Im Internet Explorer 10 wurden CSS Regions als Erstes implementiert, die entsprechende *Dokumentation* eignet sich daher zum Einstieg in die Grundlagen. Siehe http://msdn.microsoft.com/en-us/library/ie/hh673537%28v=vs.85%29.aspx/.

CSS Exclusions finden sich ebenfalls in IE10, sehen Sie also auch hier zuerst in der *Microsoft-Dokumentation* nach. Anschließend schauen Sie sich ein paar *Demos von Adobe* an. Siehe http://msdn.microsoft.com/en-us/library/ie/hh673558%28v=vs.85%29.aspx/ und http://adobe.github.com/web-platform/samples/css-exclusions/.

Im *MDN* findet sich die beste Dokumentation von Feature Queries, obwohl die API derzeit noch undokumentiert ist. Siehe https://developer.mozilla.org/en-US/docs/CSS/@supports/.

Während ich dies schreibe, ist zu Cascading Variables nur etwas im *Spezifikationsentwurf* unter http://dev.w3.org/csswg/css-variables/ zu finden.

Falls der *Vorschlag für Box Alignment* immer noch aktuell ist, während Sie dies lesen, können Sie seinen Fortschritt hier verfolgen: http://dev.w3.org/csswg/css3-align/. Den Vorschlag zu *Line Grid* finden Sie unter http://dev.w3.org/csswg/css-line-grid/.

Håkon Wium Lie und Chris Mills haben eine sehr schöne Einführung zum Seitenkonzept mit CSS geschrieben. Lesen Sie dazu ihren Artikel *Opera Reader: Paging the Web*: http://people.opera.com/howcome/2011/reader/index.html. Mehr zu Seitenvorlagen finden Sie auf dem *Adobe Web Platform-Blog* unter http://blogs.adobe.com/webplatform/2012/05/31/pagination-templates-in-css/.