Bulletproof Web Design

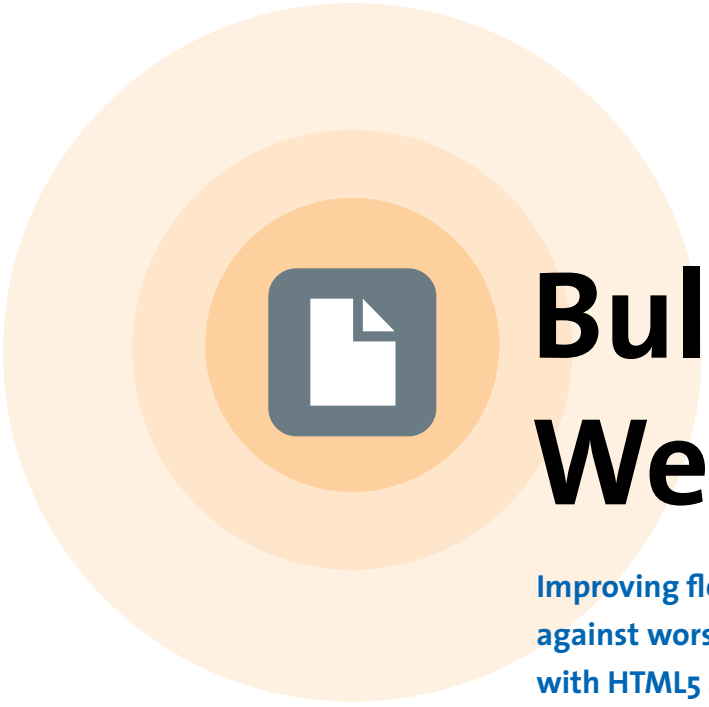
Improving flexibility and protecting against worst-case scenarios with HTML5 and CSS3

THIRD EDITION

Dan Cederholm

New Riders

VOICES THAT MATTER™



Bulletproof Web Design

Improving flexibility and protecting against worst-case scenarios with HTML5 and CSS3

Third Edition

Dan Cederholm

New Riders
VOICES THAT MATTER™

Bulletproof Web Design: Improving flexibility and protecting against worst-case scenarios with HTML5 and CSS3, Third Edition
Dan Cederholm

New Riders
1249 Eighth Street
Berkeley, CA 94710
510/524-2178
510/524-2221 (fax)
Find us on the web at: www.newriders.com
To report errors, please send a note to: errata@peachpit.com

New Riders is an imprint of Peachpit, a division of Pearson Education.

Editor: Rebecca Gulick
Copy Editor: Liz Merfeld
Technical Reviewer: Patrick H. Lauke
Proofreader: Patricia Pane
Production Coordinator and Compositor: Danielle Foster
Indexer: Valerie Haynes Perry
Cover Designers: Mimi Heft and Dan Cederholm
Interior Designers: Charlene Will and Maureen Forys

ISBN-13: 978-0-321-80835-6
ISBN-10: 0-321-80835-5

9 8 7 6 5 4 3 2 1

Printed and bound in the United States of America

For Jack & Tenley.

ACKNOWLEDGMENTS

To Michael Nolan for connecting me with New Riders as well as helping me solidify the concept of the book early on. Similarly, big thanks also go to the late, great Marjorie Baer for her guidance at the beginning of the project.

To Dave Roberts for stealing second.

To everyone at Peachpit who was involved in this book in some way, but especially Rebecca Gulick for steering the entire project, making the entire process as smooth as possible.

To Ethan Marcotte for being an excellent technical editor on the first edition. It was great to work with Ethan on this, and his comments and insight were invaluable. Thanks also to Virginia DeBolt for tech-editing the second edition and Patrick H. Lauke for tech-editing the third edition.

To my colleagues who continue to provide inspiration, sharing their techniques and knowledge, on and off line. This book wouldn't have been written if it weren't for the previous work of the following (as well as many more who I'm forgetting to mention here): John Allsopp, Dan Benjamin, Holly Bergevin, Douglas Bowman, Andy Budd, Tantek Çelik, Joe Clark, Andy Clarke, Mike Davidson, Todd Dominey, Jason Fried, John Gallant, Patrick Griffiths, Jon Hicks, Molly Holzschlag, Shaun Inman, Ryan Irelan, Richard Ishida, Roger Johansson, Jeremy Keith, Ian Lloyd, Drew McLellan, Eric Meyer, Cameron Moll, Matt Mullenweg, Mark Newhouse, Dunstan Orchard, Veerle Pieters, D. Keith Robinson, Richard Rutter, Jason Santa Maria, Christopher Schmitt, Dave Shea, Ryan Sims, Greg Storey, Jeffrey Veen, Josh Williams, and Jeffrey Zeldman.

Special thanks to Jon Hicks for creating the original 3D CSS Box Model Illustration (www.hicksdesign.co.uk/3dboxmodel/), which inspired those found in the book.

To the readers and clients of SimpleBits, you've made it possible for me to work on things I love.

To the Greater Beverly Adult Dodgeball league, for providing a necessary escape every Tuesday evening at 7:30.

To my always amazingly supportive parents, my brother Matt and his family, and the rest of my extended family and friends.

And as always, to my partner in crime, Kerry, for being the rock star wife that she is.

Lastly, thank *you* for reading.

Contents

This page intentionally left blank

Introduction

I have a confession to make. There's no such thing as a *completely* bulletproof website. Now, before you close the book and put it back up on the shelf (hopefully sticking out a bit farther than the others, thanks), allow me to explain.

Just as a police officer straps on a bulletproof vest for protection, so too can we take measures that *protect* our web designs. This book will guide you through several strategies for bulletproofing websites: improving flexibility and preparing for worst-case scenarios.

THE BULLETPROOF CONCEPT

Out in the nonvirtual world, a bulletproof vest never guarantees complete, 100% protection, but rather being bulletproof is something that's constantly strived for. You're far better off wearing a bulletproof vest than if you weren't.

The same rule applies to web design and the techniques described in this book. By increasing a page's flexibility and taking the necessary steps to ensure that it's readable in as many circumstances as possible, we're making a real difference in our work. It's an ongoing process, and one that becomes easier when utilizing web standards such as semantic HTML and CSS to construct compelling, yet adaptable, designs.

As the adoption of CSS-based layouts has become common over the past several years, it's become increasingly important to learn how to utilize CSS *well*. The goal is to harness the benefits that make the technology powerful from a design standpoint: less code, increased accessibility, and easier maintenance, to name a few.

But just using CSS and HTML doesn't necessarily mean things are automatically better. By embracing the flexibility that can be gained from separating the core content from the design, you'll be well on your way to creating better designs for all the web's citizens. But what do I mean by *flexibility* exactly?

WHY IT'S IMPORTANT

Around the time that I began thinking about the topic for this book, I realized that there are two important pieces that make up great, compelling web designs. One piece is the *visual component*—the piece that's obvious to anyone just looking at the finished page. This is a combination of the graphic design, colors, and typography the designer chose. Just visit the CSS Zen Garden (www.csszengarden.com), and it becomes obvious that compelling visual design is certainly possible and thriving when HTML and CSS are used.

The second (but equally important) piece to building a great website is the *bulletproof implementation*. It's this piece that the book will focus on: You've wisely decided to use HTML and CSS to build websites to reap all of the benefits that come along with them. And now you're ready to leverage those web standards with some ingenuity to create visually compelling websites that are also as flexible, adaptable, and accessible as possible.

As the adoption of web standards such as HTML and CSS increases rapidly, it becomes more and more important to have resources that discuss *how* these standards can be utilized and implemented in the most optimal way.

THE BOOK'S STRUCTURE

Each chapter of the book describes a certain *bulletproof guideline*. We'll start by looking at an existing design from the web, and we'll note why it isn't bulletproof. We'll then rebuild the example using HTML and CSS, with the goal of improving its flexibility and decreasing its code.

Many of these examples are specific *components* of the page, which makes it easier to talk about how they might be bulletproofed in chunks. In the final chapter, "Putting It All Together," we'll round up all of the techniques from previous chapters to create a full-page template—reminding ourselves along the way why we've chosen the bulletproof techniques, and illustrating how they all can work together in one place.

The step-by-step nature of each chapter's examples should make it easy to follow along—even if you are new to using HTML and CSS in your daily work. Along the way, I'll explain why these web standards are beneficial, and specifically how each chapter's guideline can improve a website's *bulletproofness*.

note

I'm using the term *bulletproof* partly to describe flexibility—in other words, designs for the web that can easily accommodate various text sizes and amounts of content, designs that expand or contract along with whatever is placed within them.

In addition, we can (and will) talk about flexibility from an editing, maintenance, or development view as well—improving the ease with which content is edited and code updated and maintained, while at the same time not hindering the design. This can be especially helpful and important for *internationalization* issues, where length of content can drastically vary between the various languages of the world.

And last, we'll also talk about flexibility from an environment standpoint. How will designs impact the integrity of a website's content and function? We'll make sure that what we create can adapt to a variety of scenarios, be it a web browser, screen reader, or mobile device. Designing with flexibility in mind means better interpretation by a wider range of devices and software.

THE CONTEXT OF THE BOOK'S EXAMPLES

All of the examples assume a basic page structure that surrounds them. In other words, what is shown in each chapter in terms of HTML and CSS code happens within an assumed, existing HTML5 document in between the <body> and </body>.

For instance, the basic framework for the book's examples could be set up like this:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>Page Title</title>
    <meta charset="utf-8" />
    <style type="text/css">
      ... example CSS goes here ...
    </style>
</head>
```

```
<body>
  ... example markup goes here ...
</body>
</html>
```

While the CSS is placed in the <head> of the page for convenience, it could (and probably should) be pulled out into its own file.

Using HTML5 today

All the examples in the book assume an HTML5 document, and we'll be using some elements that are new in HTML5 throughout the book. Anyone can start using HTML5 right now, simply by using the simple DOCTYPE shown earlier.

There are, however, two small steps to ensure that the new HTML5 elements are recognized and can be styled in all browsers. New HTML5 elements that didn't exist prior aren't acknowledged in Internet Explorer 8 and below. The easiest way to "tell" those browsers about the new elements is to use a simple little JavaScript shim developed by Remy Sharp: http://remysharp.com/2009/01/07/html5-enabling-script/

Simply add this line to the <head> of your documents, which will conditionally load it (from Google) for IE8 and below, enabling these fresh new HTML5 elements in IE:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Page Title</title>
  <meta charset="utf-8" />
  <style type="text/css">
    ... example CSS goes here ...
  </style>
  <!-- enable HTML5 elements in IE7+8 -->
  <!--[if lt IE 9]>
  <script src="http://html5shim.googlecode.com/svn/trunk/
  →html5.js"></script>
  <![endif]-->
</head>
```

Reset styles

In order to zero out margins, padding, and other styles that browsers typically apply by default, I'll often use a *reset stylesheet*: a batch of CSS rules that sets a consistent base for which to start styling upon. These styles come before all others, either at the top of the main stylesheet, or linked before others if using external stylesheets.

I recommend using Eric Meyer's version, which he keeps up to date here: `http://meyerweb.com/eric/tools/css/reset/`

One crucial part of the reset stylesheet when using HTML5 is to declare the new HTML5 elements as `display: block;` for older browsers that didn't know about them at the time (otherwise, you may encounter some oddness when styling them).

Here is Eric's latest version of reset (at the time of this writing), where I've highlighted the important HTML5 bits:

```
/* http://meyerweb.com/eric/tools/css/reset/
   v2.0 | 20110126
   License: none (public domain)
*/

html, body, div, span, applet, object, iframe,
h1, h2, h3, h4, h5, h6, p, blockquote, pre,
a, abbr, acronym, address, big, cite, code,
del, dfn, em, img, ins, kbd, q, s, samp,
small, strike, strong, sub, sup, tt, var,
b, u, i, center,
dl, dt, dd, ol, ul, li,
fieldset, form, label, legend,
table, caption, tbody, tfoot, thead, tr, th, td,
article, aside, canvas, details, embed,
figure, figcaption, footer, header, hgroup,
menu, nav, output, ruby, section, summary,
time, mark, audio, video {
    margin: 0;
    padding: 0;
    border: 0;
    font-size: 100%;
    font: inherit;
    vertical-align: baseline;
}
```

```
/* HTML5 display-role reset for older browsers */
article, aside, details, figcaption, figure,
footer, header, hgroup, menu, nav, section {
    display: block;
}
body {
    line-height: 1;
}
ol, ul {
    list-style: none;
}
blockquote, q {
    quotes: none;
}
blockquote:before, blockquote:after,
q:before, q:after {
    content: '';
    content: none;
}
table {
    border-collapse: collapse;
    border-spacing: 0;
}
```

So, with the HTML5 JS shim and reset styles in place as part of your framework, you'll be ready to plug in the examples that happen throughout the book. Each example assumes this setup, and that'll help us focus on the important stuff rather than repeating ourselves each time.

COMMON TERMS USED THROUGHOUT THE BOOK

There are times throughout the book that I'll refer to various browser versions by their abbreviations. For instance, it's much easier to say IE6/Win than "Internet Explorer version 6 for Windows." Here are a few browser terms we'll be using:

- IE = Internet Explorer for Windows
- WebKit = Browser engine that powers Safari and Chrome
- Mozilla = Browser engine that powers Firefox

When describing the older approaches found in the examples used for each chapter, I often refer to *nested tables* and *spacer GIF shims*. This describes the traditional techniques often used to build websites, in which tables were used to create pixel-perfect but inflexible beasts. Nesting tables inside one another made it easier to precisely align graphics and text, yet the result was a gigantic amount of code with accessibility problems galore.

The term *spacer GIF shim* refers to the use of a single transparent GIF image that's stretched to various dimensions in order to create gaps, columns, and divisions throughout a page. An *unbulletproof* website will have these littered throughout its markup, adding code and creating a maintenance nightmare.

But there are better ways of accomplishing the same visual goal using lean, meaningful markup and CSS. By embracing these web standards, we can still create compelling designs that at the same time are flexible and ready for whatever situation is thrown at them. This is *Bulletproof Web Design*.

This page intentionally left blank

Creative Floating

Use floats to achieve grid-like results.

In previous chapters, we started each bulletproof approach by first deciding how the component we're dealing with should be structured—that is, what markup is most appropriate for the content being displayed? This, I believe, is an important step when building any website. With a goal of selecting elements that convey the most *meaning* to what they surround, you hope to be left with enough "style hooks" to apply a compelling design with CSS. But more importantly, choosing the best markup for the job means the content of the page has the best chance of being understood properly across the widest possible range of browsers and devices. It's the foundation and should convey a clear message, regardless of the design applied on top of it via CSS. Additionally, if we take time to consider optimal structure (meaningful HTML) and design (CSS) *separately*, changing that design later on down the road becomes less like finding a needle in a haystack and more like changing the slipcover on a sofa.

Settling on an optimal structure doesn't have to limit the way it's displayed, however. As we'll explore in this chapter, the creative use of the `float` property can give us grid-like results—with a *fraction* of the code we'd need with the nested-tables approach. By paring down the markup to its barest essentials, we make it easier for browsers, software, and all devices to read our content—at the same time making it easier for other designers and developers to modify and edit these components.

We start this chapter off by rebuilding a box containing multiple pairings of an image, title, and description. This is a common layout requirement found on sites all over the web, and one that we can handle elegantly using minimal markup and CSS.

A Common Approach

Often found on sites to display teasers to articles, products, files, and so forth, the image/title/description package should look familiar to you. A related image is usually aligned to one side against the title and quick description (Figure 4.1).

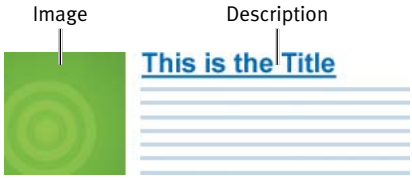


Figure 4.1 Image, title, and description "packages" like this one are found throughout the web.

There might also be several of these pairings in a row, each pointing to an article, product, or other destination. Historically, one might use a `<table>` to structure all of this, using spacer GIFs to control white space and gutters between the items (Figure 4.2).

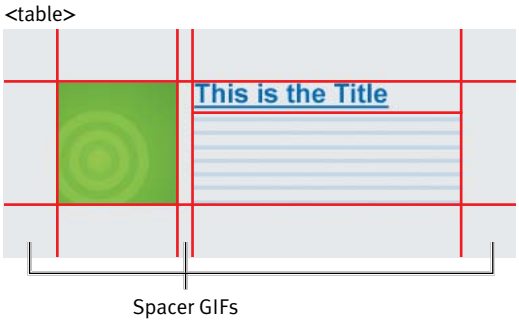


Figure 4.2 Tables and spacer GIFs may be used to space and position the items.

Figure 4.3 The Furniture Shack home page features a box of "teasers."

Some could argue that what we're dealing with here *is* tabular data (think spreadsheets, calendars, statistics), and I'm not here to debate the use of tables. What we *will* do is use a particular component from a popular real-world site as a guideline, which we'll then reconstruct using *far* less markup and CSS to achieve table-esque results. In the end, we'll toss out extraneous markup and unnecessary images to create something more flexible, accessible, and manageable from an editing viewpoint.

Figure 4.3 shows the component from Furniture Shack (a fictitious online merchant of fine home furnishings) that we'll reconstruct. As you can see, it's a bordered box containing three "teasers" to a variety of products available from the Furniture Shack stores.

Each "teaser" contains a product image, title, and short description. Both the style and layout fit the look of the company's stores and catalogs—so, well done there.

Under the hood, this box is built using a series of nested tables and spacer GIFs, and it's worth pointing out that the title text is served as an image. The code required to render this particular layout could easily fill a beautiful cherry-stained chest of drawers (with brushed-nickel knobs).

Why It's Not Bulletproof

For this particular example, the amount of code required could be reason enough for considering a better approach (Figure 4.4). Reducing the code will not only cut down on file size (which in turn will reduce required server space and speed up the downloading of pages), but it will also make the editing of the component from a production standpoint far easier. When we take the time to choose the best markup for the task at hand, the simplified results will be easier for servers *and* site editors to read and understand. Think of it as flexibility in terms of maintenance as well.

Because of the code bloat, the common approach also scores low in terms of accessibility to a wide range of software and devices. Accessing the rigid construction of nested tables and spacer GIF shims that are used to lay out the design with anything but a standard web browser could certainly prove to be trying for any user. As we'll discover, deflating the code bloat and increasing the accessibility doesn't have to compromise the design.

```
spacer.gif" width="1" height="3" alt="" border="0"><br><a href="http://ww2.furnitureshackshack.com/cat/
themeindex.cfm?cid=thmgor&altsrc=shpcfur%7Crshop" onMouseOut="MM_swapImgRestore()"
onMouseOver="MM_swapImage('editoeg3','','http://a451.g.akamai.net/7/451/1713/0001/image2.styleinamerica.com/
fsimgs/images/bld-20050401-004/common/arr_right45_white.gif',1)">new lighting<img src="http://
a451.g.akamai.net/7/451/1713/0001/image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/
spacer.gif" width="3" height="1" alt="" border="0"><img src="http://a451.g.akamai.net/7/451/1713/0001/
image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/arr_right44_white.gif" width="4"
height="7" alt="" name="editoeg3" border="0"></a><br><img src="http://a451.g.akamai.net/7/451/1713/0001/
image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/spacer.gif" width="1" height="3" alt=""
border="0"><br><a href="http://ww2.furnitureshackshack.com/view.cfm?pg=body/orgguide"
onMouseOut="MM_swapImgRestore()" onMouseOver="MM_swapImage('editoeg4','','http://a451.g.akamai.net/7/451/
1713/0001/image2.styleinamerica.com/fsimgs/images/bld-20050401-004/common/arr_right45_white.gif',1)">Our
lamps look great and even work.  Lightbulb not included.<img src="http://a451.g.akamai.net/7/451/1713/0001/
image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/spacer.gif" width="3" height="1" alt=""
border="0"><img src="http://a451.g.akamai.net/7/451/1713/0001/image2.furnitureshackshack.com/fsimgs/images/
bld-20050401-004/common/arr_right44_white.gif" width="4" height="7" alt="" name="editoeg4" border="0"></
a><br><img src="http://a451.g.akamai.net/7/451/1713/0001/image2.furnitureshackshack.com/fsimgs/images/bld
-20050401-004/common/spacer.gif" width="1" height="2" alt="" border="0"><br></font></a></td>
                    </tr>
                </table></div><div align="center">
<table border="0" cellpadding="0" cellspacing="0" width="199">
<tr>
<td colspan="2"><img src="http://a451.g.akamai.net/7/451/1713/0001/image2.furnitureshackshack.com/fsimgs/
images/bld-20050401-004/common/spacer.gif" width="1" height="24" alt="" border="0"></td>
</tr>
<tr>
<td valign="top" width="108"><table border="0" cellpadding="0" cellspacing="0"><tr><td><br><font
face="Verdana, Geneva, Arial, Helvetica" size="1" color="666666" class="text"><a href="http://
ww2.furnitureshackshack.com/cat/roomindex.cfm?cid=romliv" onMouseOut="MM_swapImgRestore()"
onMouseOver="MM_swapImage('edit5sss','','http://a451.g.akamai.net/7/451/1713/0001/image2.styleinamerica.com/
fsimgs/images/bld-20050401-004/common/arr_right45_white.gif',1)">sofas that are red<img src="http://
a451.g.akamai.net/7/451/1713/0001/image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/
spacer.gif" width="3" height="1" alt="" border="0"><img src="http://a451.g.akamai.net/7/451/1713/0001/
image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/arr_right44_white.gif" width="4"
height="7" alt="" name="edit5sss" border="0"></a><br><img src="http://a451.g.akamai.net/7/451/1713/0001/
image2.furnitureshackshack.com/fsimgs/images/bld-20050401-004/common/spacer.gif" width="1" height="5" alt=""
border="0"><br>
```

Figure 4.4 Avoid drowning in a sea of code.

A Bulletproof Approach

To simplify the structure of this box, we'll strip away the tables in favor of minimal markup. We'll then turn to our friend CSS to replicate the grid-like layout of the *teasers* (the image/title/description grouping), while still creating a compelling design.

As always, to get things started, let's decide how best to structure the box with markup.

THE ENDLESS CHOICES FOR MARKUP

In assessing what we need in terms of markup for this design, let's refamiliarize ourselves with our goal by sketching out what we need in the way of structure.

Figure 4.5 illustrates the basic framework we're dealing with. We'll need an outer container for the box to hold everything and create the border. Inside, the three teasers each contain an image floated to one side, alongside a title and short description.

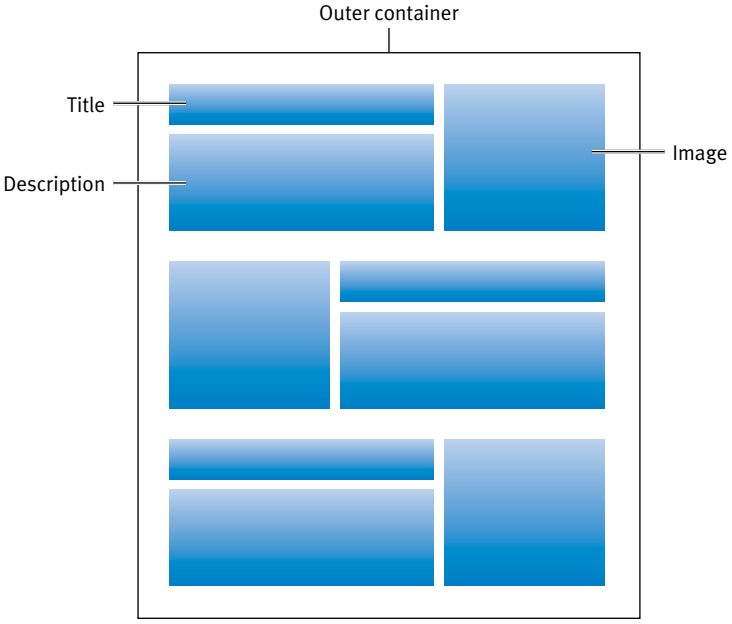


Figure 4.5 This wireframe of the structure will help us lay out our example.

Because I know what we'll be up against further on, I know we need an element that also *surrounds* each teaser. This element will group each image, title, and description together as a unit. And semantically, this makes sense as well, isolating each discrete chunk (or teaser) with a containing element.

With that said, we can weigh our options in terms of the markup we choose here. As in most things web design, there are no *wrong* choices—only choices and *better* choices.

USING DEFINITION LISTS

Definition lists are underused in my opinion, especially for applications that aren't an obvious title and description pairing (as they are commonly used for). A definition list consists of an outer `<dl>` element, with any number of definition terms `<dt>` and descriptions `<dd>`:

```
<dl>
  <dt>This is the Term</dt>
  <dd>This is the description.</dd>
</dl>
```

In its specification for definition lists (http://www.w3.org/TR/html5/grouping-content.html#the-dl-element), the W3C hints at other uses for these elements by suggesting that a) definition lists can contain multiple terms and/or definitions and b) definition lists can also be used for "...terms and definitions, metadata topics and values, questions and answers, or any other groups of name-value data." Or also, for example, a dialogue that could be marked up like so:

```
<dl>
  <dt>Younger Cop</dt>
  <dd>And was there anything of value in the car?</dd>
  <dt>The Dude</dt>
  <dd>Oh, uh, yeah, uh... a tape deck, some Creedence tapes,
  ➞and there was a,  uh... uh, my briefcase.</dd>
  <dt>Younger Cop</dt>
  <dd>[expectant pause] In the briefcase?</dd>
  <dt>The Dude</dt>
  <dd>Uh, uh, papers, um, just papers, uh, you know, uh,
  ➞my papers, business  papers.</dd>
  <dt>Younger Cop</dt>
  <dd>And what do you do, sir?</dd>
  <dt>The Dude</dt>
  <dd>I'm unemployed.</dd>
</dl>
```

It's b) that I (and other designers) have taken to heart, using definition lists for a variety of markup applications to provide a more meaningful and clearly organized structure.

So, with that in mind, I've chosen to use a *series* of definition lists to structure each image, title, and description.

THE MARKUP STRUCTURE

To make things more interesting (although I suppose furniture *can* be interesting), I'll be swapping out the Furniture Shack content for something of my own, featuring a few photos from a past trip to Sweden. Each tease will consist of a definition list containing the title as the definition *term* and the image and description as... well, *descriptions* of that title. As mentioned earlier, we'll also need an outer containing element to set a width and the border that surrounds the entire component.

With all of that mapped out, our simple markup structure looks like this:

```
<article id="sweden">
  <dl>
    <dt>Stockholm</dt>
    <dd><img src="img/gamlastan.jpg" width="80" height="80"
    →alt="Gamla Stan" /></dd>
    <dd>This was taken in Gamla Stan (Old Town) in a large
    →square of amazing buildings.</dd>
  </dl>
  <dl>
    <dt>Gamla Uppsala</dt>
    <dd><img src="img/uppsala.jpg" width="80" height="80"
    →alt="Gamla Uppsala" /></dd>
    <dd>The first three Swedish kings are buried here,
    →under ancient burial mounds.</dd>
  </dl>
  <dl>
    <dt>Perpetual Sun</dt>
    <dd><img src="img/watch.jpg" width="80" height="80"
    →alt="Wristwatch" /></dd>
    <dd>During the summer months, the sun takes forever to
    →go down. This is a good thing.</dd>
  </dl>
</article>
```

We've given the outer `<article>` (a new HTML5 element) container an `id` of `sweden`, and inside we have three definition lists, each containing a title, followed by an associated image and short description. At this point, you may be wondering why we chose to use three separate `<dl>`s as opposed to one big list. The reasoning here will be revealed a bit later.

SANS STYLE

Without any style applied to our markup, the structure is still apparent when viewed in a browser (Figure 4.6).

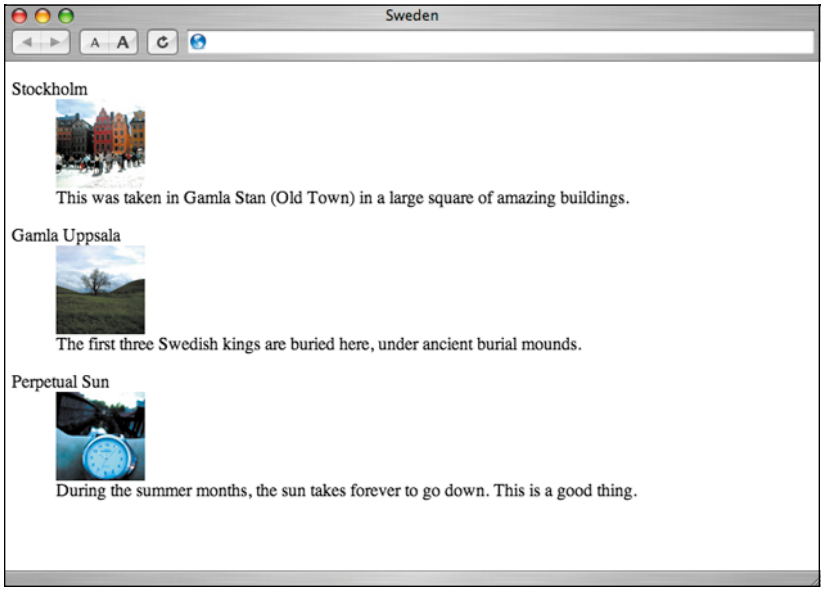


Figure 4.6 With CSS disabled, or not applied, the structure of the example is still readable and easily understood.

Typically, a browser will indent `<dd>` elements, making it easier to view the relationship between them and the `<dt>` elements that precede them. Because we've chosen lean, simple markup, any device or browsing software should have no problems whatsoever understanding what we're delivering here. But it doesn't exactly look the way we want it to yet. Let's start adding some style.

STYLING THE CONTAINER

To begin, let's add a declaration that will set a width and blue border around the entire list. We'll add these styles to the outer `<article>` previously marked with `id="sweden"`.

note

A default font of `Arial` has been set on the entire page, with `font-size` set using the absolute-size keyword `small`.

```
#sweden {
  width: 300px;
  border: 2px solid #C8CDD2;
  }
```

By setting a `width` of `300px` and a colored border around our entire box, we end up with the results shown in Figure 4.7.

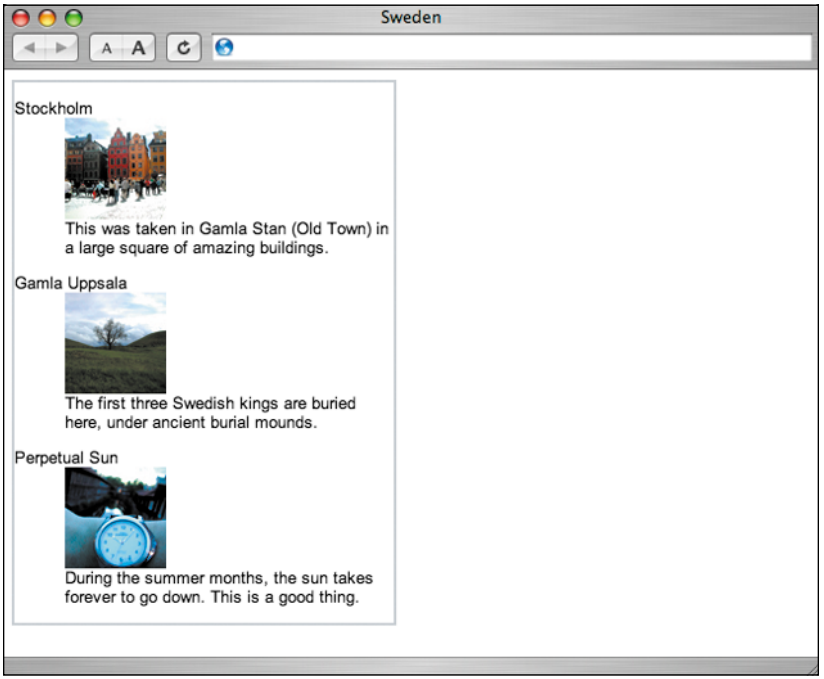


Figure 4.7 Setting a width of 300 pixels contains everything within the `<article>`.

IDENTIFYING THE IMAGE

To make things easy to control later on, one step we need to take before going any further is to add a `class` to each `<dd>` element that holds the image. Because we'll float the image (and not the second `<dd>` that holds the description text), we need a way to uniquely identify that element in the markup so that we may later apply style to it with CSS:

```
<article id="sweden">
  <dl>
    <dt>Stockholm</dt>
    <dd class="img"><img src="img/gamlastan.jpg" width="80"
    →height="80" alt="Gamla Stan" /></dd>
    <dd>This was taken in Gamla Stan (Old Town) in a large
    →square of amazing buildings.</dd>
  </dl>
  <dl>
    <dt>Gamla Uppsala</dt>
    <dd class="img"><img src="img/uppsala.jpg" width="80"
    →height="80" alt="Gamla Uppsala" /></dd>
    <dd>The first three Swedish kings are buried here,
    →under ancient burial mounds.</dd>
  </dl>
  <dl>
    <dt>Perpetual Sun</dt>
    <dd class="img"><img src="img/watch.jpg" width="80"
    →height="80" alt="Wristwatch" /></dd>
    <dd>During the summer months, the sun takes forever to
    →go down. This is a good thing.</dd>
  </dl>
</article>
```

With each `<dd>` that contains the image flagged with a `class="img"`, we're now prepared to move on.

APPLYING BASE STYLES

Let's now apply base styles for each tease, leaving only the positioning of the image to be done a bit later.

To evenly apply 20 pixels of space around the teasers as well as the inside of the box (Figure 4.8), we'll break up the margins between the containing `<article>` and the teasers themselves. First, let's apply 10 pixels of padding to the top and bottom of the containing `<article id="sweden">`. Then, we'll apply 10-pixel margins on the top and bottom of each `<dl>`, as well as 20-pixel margins on both the left and right of each `<dl>`.

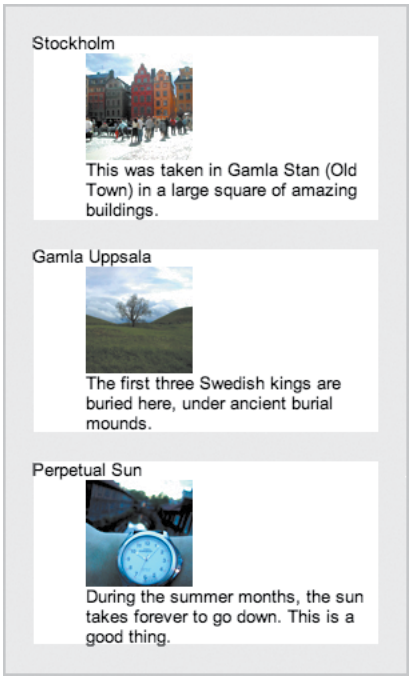


Figure 4.8 Our goal is a consistent gutter of 20 pixels (marked in gray) that flows around the inside of the box and its teasers.

```
#sweden {
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  margin: 10px 20px;
  padding: 0;
  }
```

Figure 4.9 shows the results of adding the margins and padding. We've also zeroed out any default padding that may be attached to definition lists. And the box is already looking better.

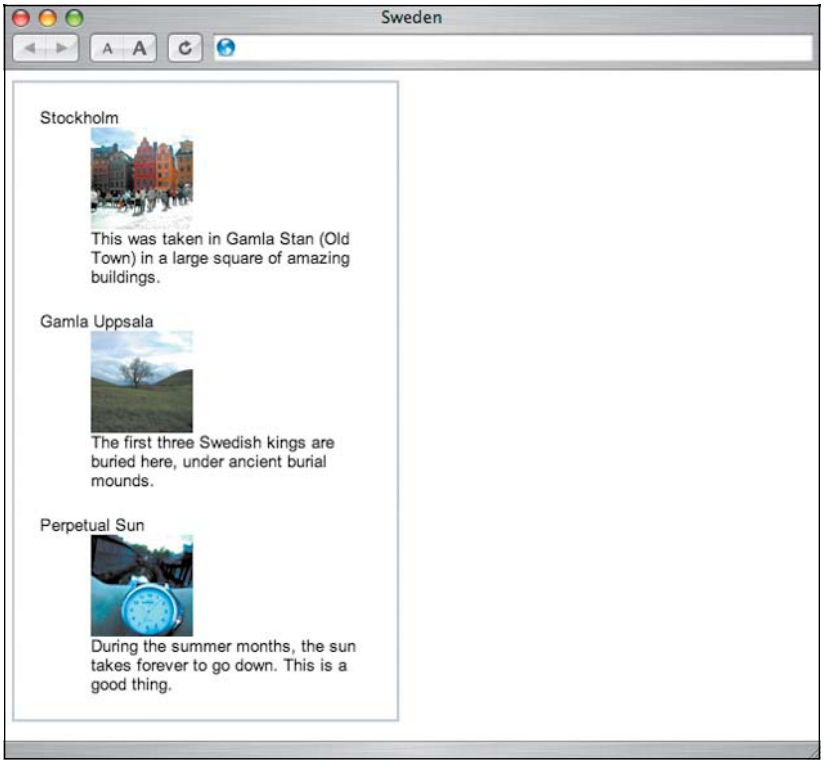


Figure 4.9 With margins and padding distributed among the elements, the box begins to take shape.

Next, let's introduce color and custom text treatment to the title of each tease by styling `<dt>` elements within our box:

```
#sweden {
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  margin: 10px 20px;
  padding: 0;
  }
#sweden dt {
  margin: 0;
```

I'm using the Safari browser throughout this example, taking advantage of Mac OS X's beautiful antialiasing of text. You could get similar results using Windows with ClearType enabled. Users of other operating systems would receive mixed results.

```
padding: 0;
font-size: 130%;
letter-spacing: 1px;
color: #627081;
}
```

Looking at Figure 4.10, you'll notice that we've increased the size of the title, added the lovely slate-blue shade, and increased space between letters by a fraction using the `letter-spacing` property, mimicking the style from the Furniture Shack example, where images were used in place of styled text.

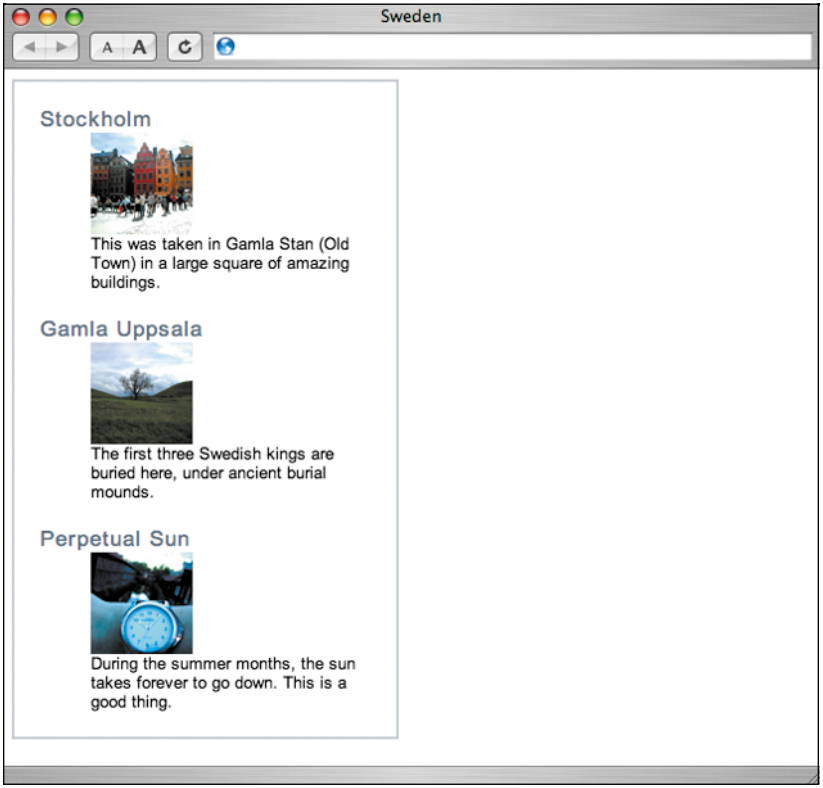


Figure 4.10 Simple text styling can do wonders for a component's design, and often eliminates the need for image-based type.

We'll also want to add a bit of style to the <dd> elements as well, matching the smaller, gray text from the Furniture Shack example:

```
#sweden {
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  margin: 10px 20px;
  padding: 0;
  }
#sweden dt {
  margin: 0;
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
#sweden dd {
  margin: 0;
  padding: 0;
  font-size: 85%;
  line-height: 1.5em;
  color: #666;
  }
```

Figure 4.11 shows the results, with the short description text now looking smaller and gray. We've also increased the `line-height` (the space between lines of text) to one and a half times the height of normal text. This lets the description breathe a bit more. Also important is the removal of the default indenting that's often applied to <dd> elements by the browser. We've overridden that by zeroing out margins and padding.

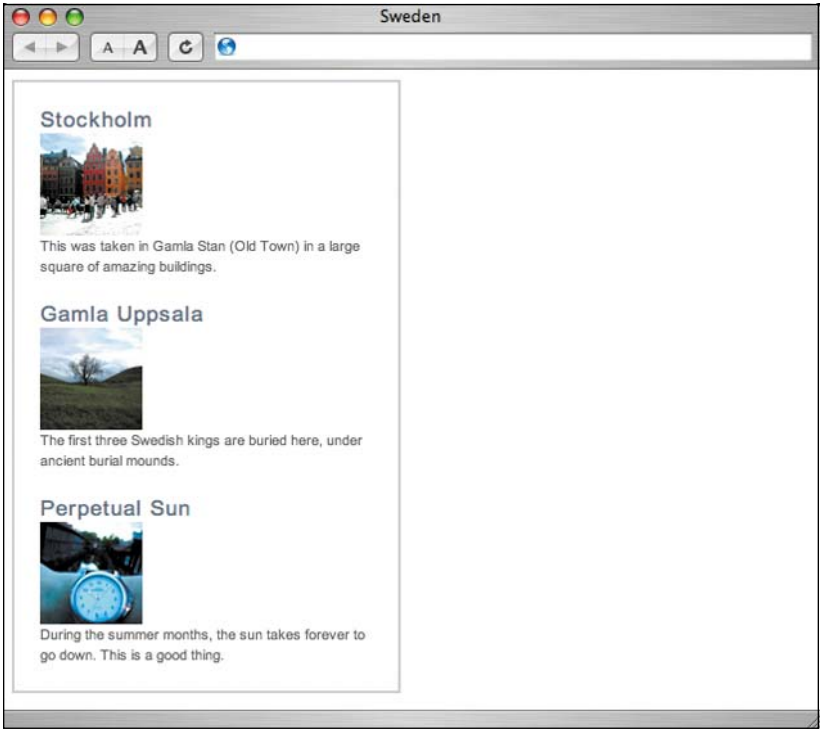


Figure 4.11 To match the Furniture Shack example, we've decreased the size of the description and made the text gray.

POSITIONING THE IMAGE

Our next challenge is to position the image to one side of both the title and description. For now, let's worry about lining things up on the *left* only. Later, we'll address how best to alternate the alignment as seen in the Furniture Shack example.

Because of the order in which things appear in the markup—title, image, then description—if we simply just float the image to the left we'll have the title always sitting *above* everything else (Figure 4.12). What we really want is the top of the image *and* title to be aligned at the same level.

Figure 4.12 With the image coming after the title in the markup, just floating it to one side leaves the title above everything.

In the past, I've swapped the order of elements to make this work, putting the image in the `<dt>` element and using `<dd>` elements for both the title and description. Because the image appeared first, floating to either side would allow us to line it up just right. But semantically, it makes far more sense to have the title be the definition term, followed by an image and text that *describe* that title (as we've done in our markup structure for this example). It takes only a little more CSS magic to have the best of both worlds: optimal markup structure and the image to one side of both the title and description.

OPPOSING FLOATS

You may remember the "opposing floats" method explained in Chapter 3, "Expandable Rows." Essentially, we used the `float` property to place two elements on opposite ends of a container. We use the same method here, allowing us to align the image to the left of both the title *and* the short description while keeping the markup in an optimal order.

Remember that we've previously tagged `<dd>` elements that contain the image with a `class="img"`—which allows us to float images within those elements to one side while keeping the descriptions in place.

So, to begin let's float `<dt>` elements *right* and images *left*:

```
#sweden {
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  margin: 10px 20px;
  padding: 0;
  }
#sweden dt {
  float: right;
  margin: 0;
```

```
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
#sweden dd {
  margin: 0;
  padding: 0;
  font-size: 85%;
  line-height: 1.5em;
  color: #666;
  }
#sweden dd.img img {
  float: left;
  }
```

By using the opposing floats method, we can position the image to the left of both the title and description, regardless of the fact that the title comes *first* in the markup (Figure 4.13).

Figure 4.13 Here we see opposing floats at work, aligning the tops of the image and title.

Notice that while we've successfully positioned the image, the description has slipped between the image and title. To fix this, we need to apply a little math to set up a grid-like layout for each tease.

Quite simply, we just need to assign a width on the <dt> elements, forcing them to span across the top of each description on their own line. To calculate this width, let's start with the total width of the box (300 pixels), minus the margins around each definition list (20 pixels times 2), minus the width of the image (80 pixels). The result is 180 pixels (Figure 4.14).

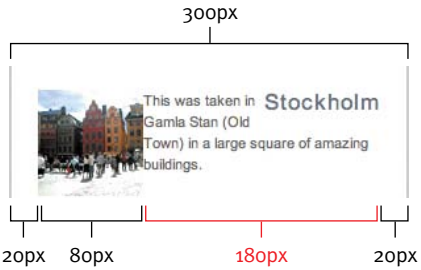


Figure 4.14 Determining the width for the <dt> elements involves a little calculation.

By simply adding a width of 180px to the declaration for <dt> elements, we ensure that things start falling into their intended places:

```
#sweden dt {
  float: right;
  width: 180px;
  margin: 0;
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
```

Figure 4.15 shows where we are at this point, having successfully positioned the image, title, and description via opposing floats.

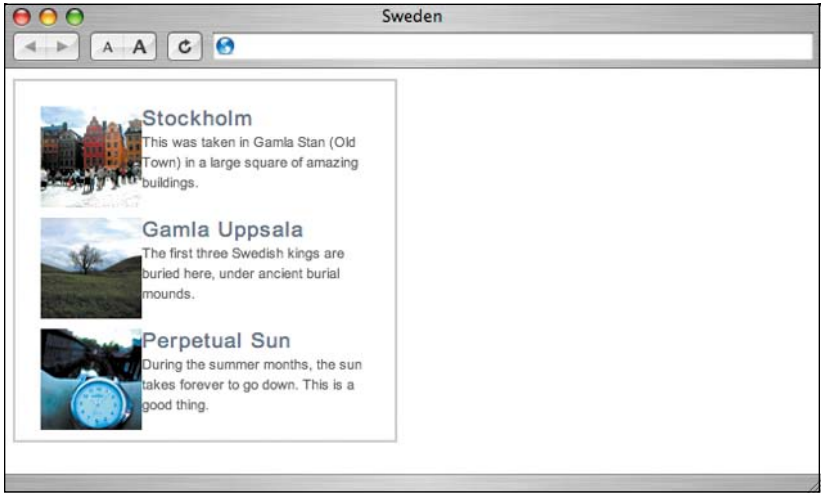


Figure 4.15 With the width assigned for <dt> elements, the items fall into place.

CLEAR THE WAY FOR ANY DESCRIPTION LENGTH

Thus far in the example, each description has been long enough to roughly meet the bottom of each image. Because of this length, we haven't yet had to worry about *clearing* the floats. To illustrate what could happen when the description is shortened, take a look at Figure 4.16. Not exactly bulletproof, is it?

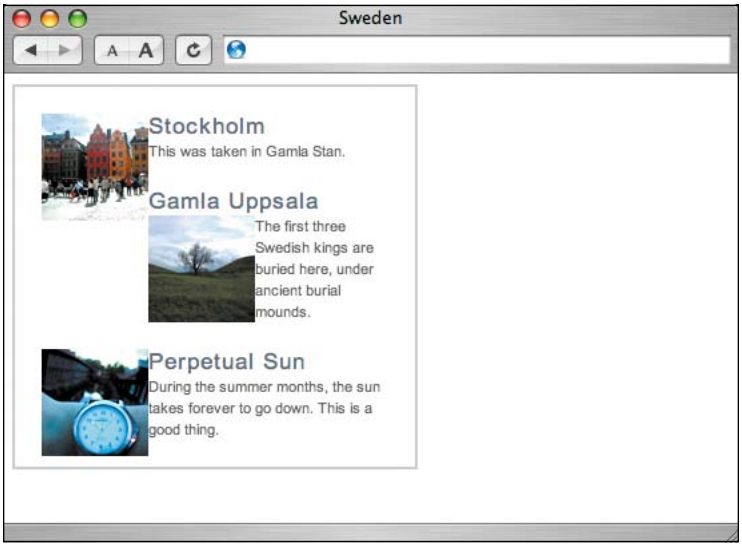


Figure 4.16 With a shorter description, the floated images can lead to undesired results.

When you are first learning how to use floats, understanding how they need to be properly *cleared* can be tricky. When an element is floated, it is taken out of the normal flow of the document and doesn't affect the vertical stacking of elements that follow it. Floated elements will always extend beyond the height of their containers. If the description happens to be long enough to meet or exceed the bottom of the floated image, then all is right with the world. But if the content next to the floated image is *shorter*, that's when you'll run into problems.

Figure 4.17 shows the outline of the definition list marked in red. You can see that the image is taller than the title and description combined in the first teaser. And since the image is floated left, the next definition list in line will attempt to wrap around it. What we need is a way to clear the floated image before going on to the next teaser. For example, in the old days one

might add <br clear="all"> to clear any previously declared floats in the markup. This works, but is rather unnecessary when we're dealing with CSS-based designs, not to mention that the clear attribute is considered invalid in recent HTML specifications. What we'd rather do is use CSS (and not markup) to clear floats, and we'll explore a few ways to do just that next.

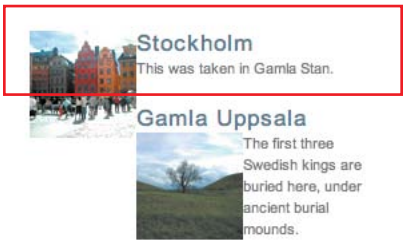


Figure 4.17 The red box shows where the <dl> containing the float really ends.

SELF-CLEARING FLOATS

There are several ways to clear floats using CSS, and to get a handle on many of them (and why problems arise), I encourage you to start off by reading CSS guru Eric Meyer's article "Containing Floats" (www.complexspiral.com/publications/containing-floats/).

I'm going to share three popular methods for *self-clearing* floats—that is, the process of applying CSS to a container that has floated elements inside it. By self-clearing the floats within, it **keeps the container independent**, regardless of what comes before or after it in the flow of the document. This is a key element of being bulletproof: If we keep "modules" (containers of various mini-layouts) independent, they stand a better chance of staying intact if moved around, changed, or edited later by either you or your client or boss. Self-clearing is essential for that modularization when you're using floats and requires no extra markup (such as <br clear="all">).

Let's look at three ways to do this, applying each method to our example:

- **The "Set a Float to Fix a Float" method** (described in Eric Meyer's article and used previously in this book). This technique often depends on what comes after the container on the page, but this cross-browser method is simple to implement.
- **The "Simple Clearing of Floats" method using the** overflow **property.** This is probably the simplest method to implement but has some possible side effects. It's described in detail at SitePoint: www.sitepoint.com/blogs/2005/02/26/simple-clearing-of-floats/.

- **The "Easy Clearing" method using generated content** (described at http://positioniseverything.net/easyclearing.html). This method requires jumping through a few hoops for Internet Explorer, but once you grasp the idea, I think it's the most solid choice.

First, we'll use the "Set a Float to Fix a Float" method and apply that to our example.

Setting a float to fix a float

Essentially, a container will stretch to fit around floated elements within it—if the container is *also* floated. So, taking Eric Meyer's advice, we want to float each <dl> left in order to force each teaser below the floated image above it. In addition, we need to float the entire containing <article>, ensuring that the border will enclose all of the floated elements within it:

```
#sweden {
  float: left;
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  float: left;
  margin: 10px 20px;
  padding: 0;
  }
#sweden dt {
  float: right;
  width: 180px;
  margin: 0;
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
#sweden dd {
  margin: 0;
  padding: 0;
  font-size: 85%;
  line-height: 1.5em;
```

```
  color: #666;
  }
#sweden dd.img img {
  float: left;
  }
```

Adding the previous rules to our example, we have properly cleared floats, with each teaser ending up below the other—regardless of description length (Figure 4.18).

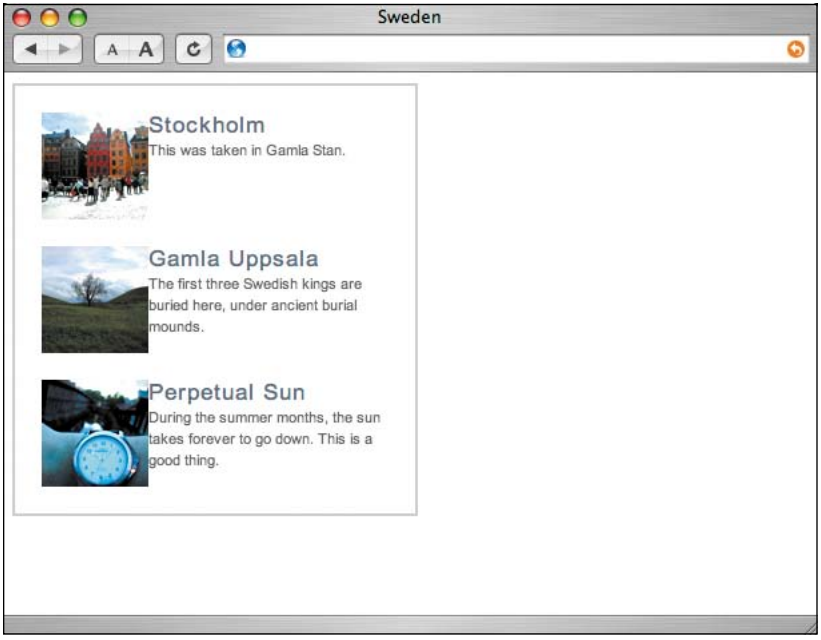


Figure 4.18 Even with a short description, we have properly cleared floats.

THE FINISHING TOUCHES

To put the final polish on this example, let's adjust the spacing between images and text, and later allow for floating the image both to the left *and* right.

To adjust the spacing between the images and text, we'll just have to add a right margin to the image, then subtract that amount from the width we've defined for the <dt> elements. While we're at it, let's also add a little framed border around each image. We can fold these additions into the master stylesheet for this example, where you can see that the CSS remains rather compact:

```
#sweden {
  float: left;
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  float: left;
  width: 260px;
  margin: 10px 20px;
  padding: 0;

  }
#sweden dt {
  float: right;
  width: 162px;
  margin: 0;
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
#sweden dd {
  margin: 0;
  padding: 0;
  font-size: 85%;
  line-height: 1.5em;
  color: #666;
  }
#sweden dd.img img {
  float: left;
  margin: 0 8px 0 0;
  padding: 4px;
  border: 1px solid #D9E0E6;
  border-bottom-color: #C8CDD2;
  border-right-color: #C8CDD2;
  background: #FFF;
  }
```

Figure 4.19 shows the results of the new styles. We've added an 8-pixel margin to the right of each floated image, as well as 4 pixels of padding and a 1-pixel border around the image itself to create a frame. Since we've added this extra width, we have to add all that up, then subtract that total from the width we previously set on <dt> elements. The math goes like this: 8-pixel right margin + 4-pixel padding on both sides + 1-pixel border on both sides = 18 pixels. So, as you can see, we've dropped the width for <dt> elements down to 162 pixels to accommodate the extra space that the image will now take up.

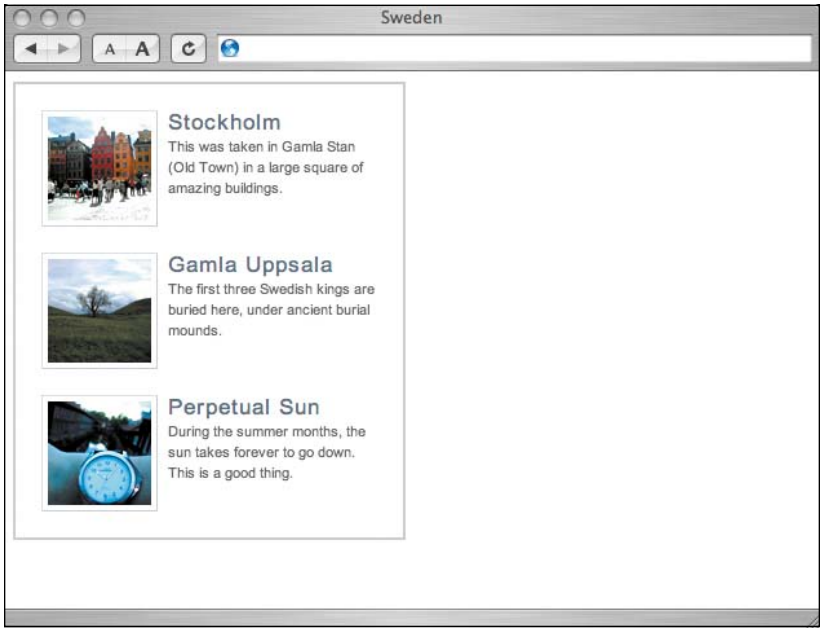


Figure 4.19 We've achieved proper spacing between images and text.

For the border around the image, we've chosen to make the right and bottom edges a slightly darker shade than the top and left. This creates a subtle three-dimensional effect on the photo, as if it's lying on top of the box (Figure 4.20).

Figure 4.20 A simple trick for creating dimension is to use darker right and bottom borders.

TOGGLING THE FLOAT DIRECTION

Another aspect of the original Furniture Shack example that we'll want to replicate is that the side to which each image floats swaps back and forth. One teaser will have the image aligned left, while another will have it aligned right. We want to build in the ability to change this at will, with a simple `class="alt"` added to the `<dl>` when a swap is desired.

First, we tag the `<dl>` that we'd like to change float direction on. We've chosen the second teaser:

```
<article id="sweden">
  <dl>
    <dt>Stockholm</dt>
    <dd class="img"><img src="img/gamlastan.jpg" width="80"
    ➞ height="80" alt=""Gamla Stan" /></dd>
    <dd>This was taken in Gamla Stan (Old Town) in a large
    ➞ square of amazing buildings.</dd>
  </dl>
  <dl class="alt">
    <dt>Gamla Uppsala</dt>
```

```
    <dd class="img"><img src="img/uppsala.jpg" width="80"
    →height="80" alt="Gamla Uppsala" /></dd>
    <dd>The first three Swedish kings are buried here,
    →under ancient burial mounds.</dd>
  </dl>
  <dl>
    <dt>Perpetual Sun</dt>
    <dd class="img"><img src="img/watch.jpg" width="80"
    →height="80" alt="Wristwatch" /></dd>
    <dd>During the summer months, the sun takes forever to
    →go down. This is a good thing.</dd>
  </dl>
</div>
```

Having added the `alt` class to the second teaser, we can now include a few rules at the end of our stylesheet that will override the default, which currently aligns the image to the left. The `alt` style will reverse the direction, floating the image on the *right*. This class could be swapped in and out at will, giving site editors easy control over the layout of the box.

The following CSS should be placed after the previous declarations we've already written:

```
/* reverse float */

#sweden .alt dt {
  float: left;
  }
#sweden .alt dd.img img {
  float: right;
  margin: 0 0 0 8px;
  }
```

Here we're telling the browser that it should do the following:

- For `<dt>` elements within a `<dl>` marked with the `alt` class, float those *left* (instead of the default right).
- Float images within the `alt` class *right* (instead of the default left).
- Change the 8-pixel margin that was to the right of the image over to the left instead.

Figure 4.21 shows the results of adding these two little declarations to the stylesheet. Because the second teaser is marked with the `alt` class, its image is aligned to the right. The idea here is that we can add or remove a simple class at any time to assign the float direction for a particular image.

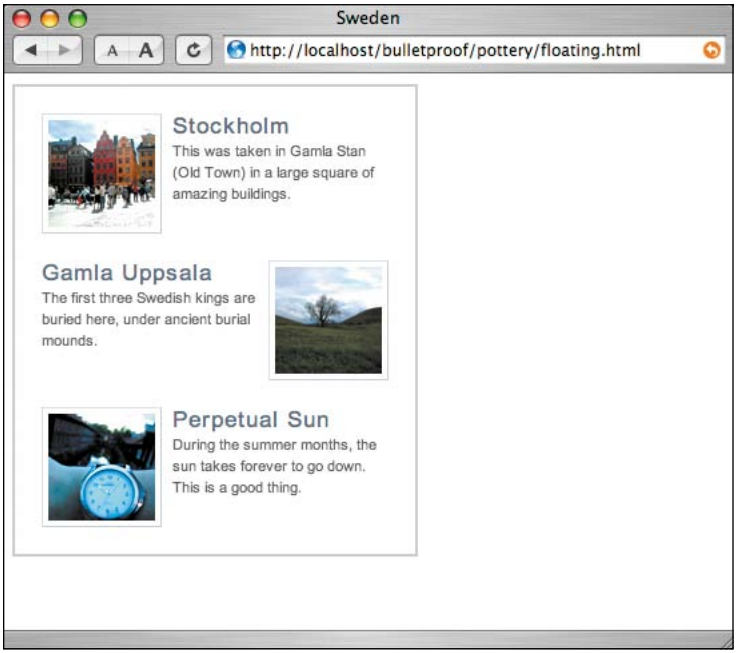


Figure 4.21 Toggling the image alignment is as simple as adding or removing the `alt` class.

THE GRID EFFECT

If we were dealing with longer descriptions (or if the user increased the text size), we'd find that the description text would wrap down around the image. That's the nature of a float: It will take up as much space as it needs to but will let content flow around it (Figure 4.22).

Figure 4.22 Longer descriptions will wrap around the floated image.

This could be the intended effect, but if a more column-like grid is what you're after, applying a margin to the description will keep the text and images away from each other.

The width of the margin that we'll add to the description should equal the width of the image, plus the padding, borders, and margin already specified between the image and description (Figure 4.23).

Figure 4.23 Adding the image width, margin, padding, and border together comes to 98 pixels.

So by adding a left margin of `98px` to all `<dd>` elements and then overriding that value to `0` for `<dd class="img">` elements (since we don't want the *image* to have a margin, yet it resides inside a `<dd>`), we'll in a sense be creating *columns* on either side.

To reverse the margins for the `alt` class when the image is floated right instead of left, we need to add another rule to our "reverse float" section at the end of the stylesheet:

```
#sweden {
  float: left;
  width: 300px;
  padding: 10px 0;
  border: 2px solid #C8CDD2;
  }
#sweden dl {
  float: left;
  width: 260px;
  margin: 10px 20px;
  padding: 0;
  }
#sweden dt {
  float: right;
  width: 162px;
  margin: 0;
  padding: 0;
  font-size: 130%;
  letter-spacing: 1px;
  color: #627081;
  }
#sweden dd {
  margin: 0 0 0 98px;
  padding: 0;
  font-size: 85%;
  line-height: 1.5em;
  color: #666;
  }
#sweden dl dd.img {
   margin: 0;
   }
#sweden dd.img img {
  float: left;
  margin: 0 8px 0 0;
  padding: 4px;
```

```
  border: 1px solid #D9E0E6;
  border-bottom-color: #C8CDD2;
  border-right-color: #C8CDD2;
  background: #FFF;
  }

/* reverse float */

#sweden .alt dt {
  float: left;
  }
#sweden .alt dd {
  margin: 0 98px 0 0;
  }
#sweden .alt dd.img img {
  float: right;
  margin: 0 0 0 8px;
  }
```

Notice that we've added a declaration that resets the margin value to `0` for `<dd>` elements that are flagged with `class="img"`. This will override the *right* margin set in the "reverse float" section farther down the stylesheet. It will also save us from repeating the override after the `#sweden .alt dd` declaration assigns a right margin that we again don't want showing up on `<dd>` elements that contain our floated image.

Figure 4.24 shows the results of the previous additions to the stylesheet. You can see that with the text size significantly increased, and/or with longer descriptions, the text and image both stick to their respective "columns," as if we'd used a table here for layout.

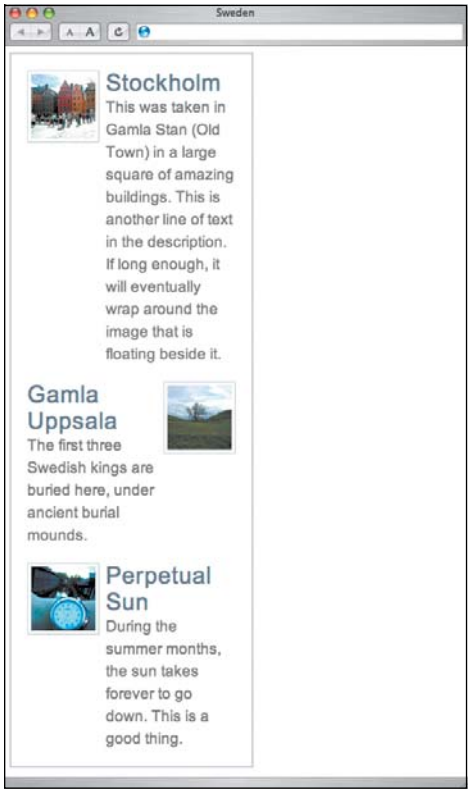


Figure 4.24 With a margin applied to the description, it's as if the text is held within columns.

AN ALTERNATE BACKGROUND

As a final touch to this example, let's trade in the solid blue border that surrounds the box for a background image that fades to white. We'll create the image in Photoshop, at a width of 304 pixels (300 pixels plus a 2-pixel border on both sides).

Figure 4.25 shows the completed image, which we created by filling a bordered box with a lighter shade of blue and then using the Gradient tool (Figure 4.26) to fade white to transparent from bottom to top.

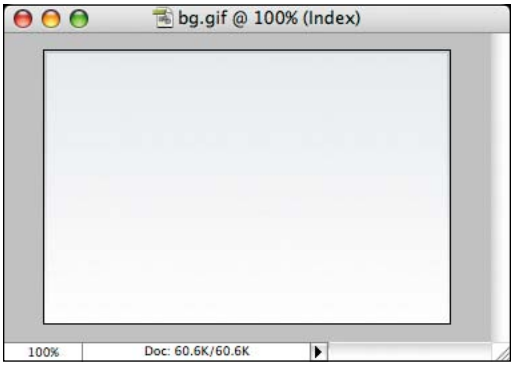


Figure 4.25 We created a blue background by using the Gradient tool.

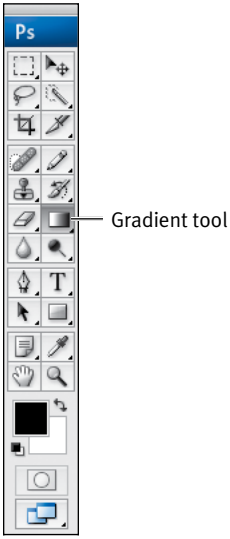


Figure 4.26 You'll find the Gradient tool in the Tools palette in Photoshop.

To reference this image in our stylesheet, we just adjust the declaration for the main containing `<article>`:

```
#sweden {
  float: left;
  width: 304px;
  padding: 10px 0;
  background: url(img/bg.gif) no-repeat top left;
  }
```

We've adjusted the width of the container from 300 pixels to 304 pixels to account for the loss of the 2-pixel border (which is now part of the background image), and we've aligned the fade top and left. Because it fades to white (the background color of the page) and is aligned at the top, we need not worry about what's contained *in* the box; it will accommodate any height, with its contents just spilling out of the fade. Another plus is that I happen to think it just looks cool (Figure 4.27).

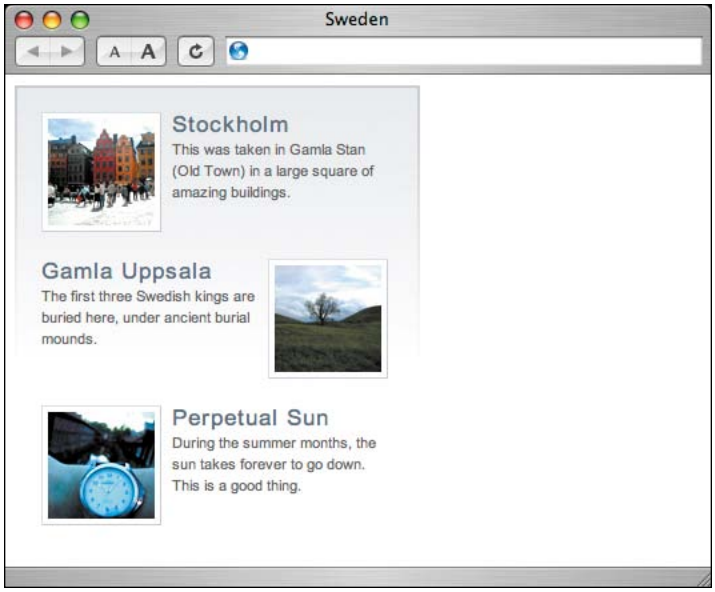


Figure 4.27 Here is our completed, bulletproof example.

If we zoom in on the top image, you can see that the padded frame remains white on top of the blue background (Figure 4.28). You may remember that we assigned a `background: #FFF;` in addition to `4px` of padding on the floated images to accomplish this. If we hadn't specified a background color here, then the blue fade would show through the image's frame.

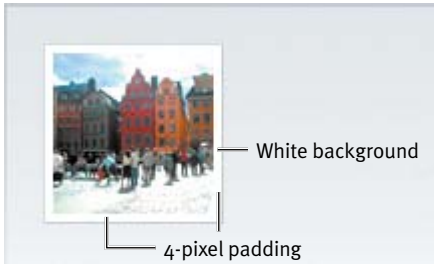


Figure 4.28 Combining padding and a background color creates a frame around the image.

APPLYING A BOX-SHADOW

As an additional detail, let's apply a CSS3 box-shadow to each thumbnail image. The box-shadow property is a wonderfully flexible way of adding shadows to elements without the need to add extra markup or images to the design. The support for box-shadow is also decent in recent browsers—but again it's one of those nonessential treatments that is often unmissed in browsers that don't support it.

To add a subtle drop-shadow to each image in our example, we need to add just a few rules to the following declaration:

```
#sweden dd.img img {
  float: left;
  margin: 0 8px 0 0;
  padding: 4px;
  border: 1px solid #D9E0E6;
  border-bottom-color: #C8CDD2;
  border-right-color: #C8CDD2;
  background: #FFF;
  -webkit-box-shadow: 1px 1px 3px rgba(0,0,0,.12);
  -moz-box-shadow: 1px 1px 3px rgba(0,0,0,.12);
  box-shadow: 1px 1px 3px rgba(0,0,0,.12);
  }
```

Here we're adding a box-shadow with the appropriate vendor prefixes for WebKit and Mozilla browsers. As usual, we always end with the non-prefixed version of the property for future proofing.

The syntax for box-shadow takes a top and left coordinate for where the shadow should begin from in relation to the element (in this case 1px from the top and 1px from the left). The third value in the rule tells the browser how much blur to apply to the shadow (in this case 3px of blur). Finally, we define the color of the shadow. Here we're using an RGBA value, which will allow us to set the color as partially transparent, letting whatever background is behind the thumbnail blend in. I almost always use RGBA values when setting box-shadows. In this particular case, we are setting `0,0,0` (the RGB value of black) at `.12` (or 12%) opacity.

Figure 4.29 shows the results in Safari after adding the box-shadow rules. For browsers that don't yet support box-shadow (e.g., IE8 and below), they'll safely ignore those rules and not display the shadow. No harm done.

The nice thing about using box-shadow for drop shadows, rather than attaching images, is that the shadows don't require space to be allotted in the layout. In other words, the shadows will bleed out into gutters or other areas of the site that aren't explicitly accounted for in the stylesheet. Historically, you'd need to create the space where the shadow appears with padding, or extra markup with specific widths. With box-shadow, it either adds the shadow where it needs to, or doesn't (if the browser doesn't support the property). Worry-free bulletproofing.

Figure 4.29 A zoomed-in view of the thumbnail image with subtle box-shadow rendering in Safari.

MORE FLOAT-CLEARING FUN

We've just finished the example using the "float to fix a float" method that's been used previously in the book. We floated each <dl> in order to clear the opposing floats that were contained within.

This is a simple concept to grasp—and an easy method to implement cross-browser. But it's somewhat reliant on a few things: We must set a width on the container (so that the floated containers stack vertically), and we must anticipate what comes before or after the container of floats (since we could run into issues with the floated *container* needing to be cleared as well, thus starting a vicious, never-ending cycle).

So while floating a container to clear floats within it can work in certain circumstances, there are other methods that are consistent regardless of the scenario. Let's take a look at a few more options.

Simple clearing of floats using the overflow property

Applying the `overflow` property on a container will self-clear any floats within it (see Alex Walker's article at SitePoint: `www.sitepoint.com/blogs/2005/02/26/simple-clearing-of-floats/`). The approach is simple and easy to implement, although it's not obvious that it will work under most circumstances. But it does.

Using our example, if we removed the floats from each `<dl>` and replaced them with `overflow: auto`, we'd be achieving the same goal.

Here's the original declaration:

```
#sweden dl {
  float: left;
  width: 260px;
  margin: 10px 20px;
  padding: 0;
  }
```

And here it is with the `overflow` trick to clear floats instead:

```
#sweden dl {
  overflow: auto;
  width: 260px;
  margin: 10px 20px;
  padding: 0;
  }
```

Now in most circumstances, the `overflow` trick will likely work out fine—but there are situations where it could become problematic. We've used the value `auto`, which could trigger scrollbars around the element should its contents be wider than its specified width (`260px`). So that's one possible scenario. You could also specify `overflow: hidden;` instead of `overflow: auto;`. The `hidden` value will (you guessed it) hide its contents should they exceed the container's width.

If you're positive that neither of those scenarios will happen, then perhaps `overflow` is worth a shot. As for me, I'm more likely to want a solution that I don't have to worry about in the future—and that's just what we'll get with the next method.

EASY CLEARING USING GENERATED CONTENT

The most robust, reliable solution for self-clearing I've found is documented in an article at Position Is Everything: "How to Clear Floats Without Structural Markup" (http://positioniseverything.net/easyclearing.html). The idea is rather clever: It uses the :after pseudo-element in CSS (www.w3.org/TR/CSS21/selector.html#before-and-after) to insert a period after the container of floats. The clear: both rule is also added to clear all floats, and then the period is hidden. Let's take a look at the declaration in action:

```
#sweden dl {
  width: 260px;
  margin: 10px 20px;
  padding: 0;
  }
#sweden dl:after {
  content: ".";
  display: block;
  height: 0;
  clear: both;
  visibility: hidden;
  }
```

For the main #sweden dl declaration, we don't need to add float or overflow as we did for the previous methods. It's the next declaration that does the float-clearing magic, so let's break it down and figure out exactly what is happening here.

We're essentially saying, "Put a period after each definition list (content: "."; will do that), clear all floats that precede them (clear: both;), and then make sure the period is hidden (height: 0; and visibility: hidden;)." Pretty crafty, eh?

By using the :after pseudo-element to self-clear any floats, we've done so without having to float the container, and we don't have to worry that overflow may cause issues down the road.

If only this worked in *cough* Internet Explorer. Yes, that's right, IE6 and 7 *don't* support the :after pseudo-element (ditto for :before). But not to worry—fortunately, there are additional rules we can target to those specific

If you're curious, `height: 1%;` is actually triggering something in IE called `hasLayout`. You can read a thorough (but lengthy) article titled "On Having Layout" (`www.satzansatz.de/cssd/onhavinglayout.html`) if you're interested in understanding the inner workings. Tread softly and pour yourself an extra large coffee before diving into this article.

versions of IE that self-clear floats just as confidently. And the good news is that IE8+ supports the `:after` and `:before` pseudo-elements, so no additional trickery is required for the future.

Autoclearing in IE6

Conveniently, IE/Win will do its own "autoclearing" on containers as long as a dimension is applied to it. This isn't correct behavior, of course, but we'll use this spec deviation to our advantage. Historically called the "Holly Hack"(named after Holly Bergevin, the hack's author), it involves setting a `height: 1%;`, targeted specifically to IE and IE only, that forces the container to expand around its contents (even floated elements). If the height of the container's contents exceeds 1% (which it almost always will), that's OK—it will expand as needed, thus ignoring the `height` rule. Again, this isn't correct behavior in terms of the spec, but it's an inconsistency that actually helps us solve this autoclearing problem (among others).

The hack uses the `* html` selector (which precedes the declaration) and is read only by IE/Win version 6 (we'll tackle IE7 in just a moment):

```
* html #sweden dl { height: 1%; }
```

That one declaration will autoclear any floats in each definition list, and by using the `* html` hack, you will target IE6 *only*. Other modern browsers will ignore it, likely because it's a nonsensical selector—there's never an element *before* the `<html>` element.

So, by offering the `:after` declaration for modern browsers that recognize it (Mozilla, Firefox, Safari, and so on), as well as the Holly Hack (for IE6), we have a majority of the browser market share covered with solutions that self-clear in a solid, reliable way. Ideally, we'd keep the Holly Hack (and any other IE-specific CSS) quarantined in its own stylesheet (we'll talk more about why later in Chapter 9).

Solving the IE7 problem

IE7 was released with significantly better standards support—*a lot* was fixed, and we thank the developers for that. But support for *everything* that browsers like Firefox, Safari, and Opera have provided for years didn't materialize—which makes a few things a little tricky. This easy-clearing method is one of them.

IE7 fixed the `height: 1%;` bug. Where IE6 would expand a container to shrink-wrap its contents—even when that exceeded a dimension specified with CSS—IE7 will now *correctly* honor that dimension. This is now proper behavior, so we can't blame the IE7 developers for fixing this. And like other, more standards-aware browsers, IE7 also now ignores declarations that begin with the `* html` selector, which is also proper behavior.

The problem, though, is that IE7 doesn't support the `:after` pseudo- element. And so by fixing the `height: 1%` bug and not supporting `:after`, the IE7 developers ensured that we're stuck in terms of getting the browser to auto-clear floats using one of these methods.

To address this issue, enter this bit of code:

```
*:first-child+html #sweden dl { min-height: 1px; }
```

Setting a `min-height` on a container in IE7 also expands a container in the same fashion that adding a `height` in IE6 does. That wacky-looking selector that precedes `#sweden dl` (`*:first-child+html`) is the piece that targets IE7 and IE7 *only*.

Hooray! Now we have the three pieces in place that will self-clear things in the most popular browsers. Let's take a look at all of them together in one place:

```
#sweden dl:after { /* for browsers that support :after */
  content: " .";
  display: block;
  height: 0;
  clear: both;
  visibility: hidden;
  }
* html #sweden dl { height: 1%; } /* for IE5+6 */
*:first-child+html #sweden dl { min-height: 1px; } /*
→ for IE7 */
```

Again, ideally the patches that target IE specifically would be placed in a separate stylesheet, kept separate so as to avoid tainting the clean code (more about that in Chapter 9, "Putting It All Together"). But with these three declarations, we have a solid, reliable way of self-clearing floats. The code may seem a bit verbose at first, but once you start using these rules consistently, you'll find that they become indispensable for creating flexibility without concern that your floats will fall apart in the future.

Combining selectors to save code

You can also combine selectors that share this self-clearing code by building a single declaration instead of repeating them for each container. For instance, suppose you are also self-clearing floats found in the header of the document:

```
#header:after,
#sweden dl:after { /* for browsers that support :after */
  content: ".";
  display: block;
  height: 0;
  clear: both;
  visibility: hidden;
  }

* html #header,
* html #sweden dl { height: 1%; } /* for IE6 */

*:first-child+html #header,
*:first-child+html #sweden dl { min-height: 1px; } /*
→ for IE7 */
```

As you're constructing your layout, you simply add elements to these three declarations as necessary, thus saving you from repeating the code for each.

Choosing what works for you

Now that we've looked at three methods for self-clearing, I'd like to point out that (as with almost everything in web design) there's no *one* way that is correct 100% of the time. After experimenting with them all, you'll find that one method might work better than the others for you, depending on the situation.

I like the easy-clearing method because of its robustness in just about any scenario, but I'll often throw `overflow: hidden;` on a container when *initially* building a layout, which is quick and easy to remember while prototyping. Later, I'll do a quick search for that rule and build the three declarations that make up the easy-clearing method.

Whatever method you choose, the important thing to remember is that **self-clearing can be a powerful tool** that maintains the flexibility that floats can bring to layout while keeping components of the page independent.

Why It's Bulletproof

After all of the CSS sorcery we've applied to this example, we may have long forgotten why we went down this path to rebuild this box of image, title, and descriptions. But in choosing a lean, semantic markup structure over the lines and lines of code that it takes to create this using tables and spacer GIFs, we're left with a compact and flexible unit—flexible in part because of its ease in updating, adding, or removing the content. The simplicity of the code also makes it easier for browsers, software, and even site editors to read.

You could also chalk this up as a big win for database-driven sites, where a uniform and more predictable markup structure could be reused more easily. Instead of worrying about whether the image comes before the title and description in the markup (when dealing with tables), the structure remains consistent and easily repeated in dynamic templates.

In choosing to use CSS to structure our grid-like layout, we've removed presentational code and graphic titles from the component, and in turn have gained flexibility in image placement and title/description size, style, and content.

Summary

By working through the construction of this example, my hope is that it will help make a case for creatively using floats to achieve grid-like effects. By understanding how to clear floats, you gain access to a wide range of possibilities without having to add extra markup. There are certainly times when using a `<table>` is appropriate (for tabular data), but even using compact and meaningful markup doesn't mean the design has to suffer. We can serve markup that a wider range of devices and software can easily understand, and use CSS to do the rest.

Here are a few points to consider:

- Choose optimal markup from the start. You can always adjust as needed, but try to get by with the most compact, most meaningful structure when you are just beginning.
- Consider the "opposing floats" method for aligning items on either side of each other, regardless of where they appear in the markup. In our example, the title came before the image, but we wanted both to line up vertically. Opposing floats to the rescue!

- Be aware of clearing floats: Test various amounts of content and text sizes to ensure your intended layout doesn't break down.
- Use the box-shadow property to handle nonessential drop shadows on elements to avoid extra markup, images, and layout adjustments.
- Experiment with the different methods of self-clearing floats to maintain flexibility and components with independence.

Index

C

D

E

J

K

L

M

N

O

P

R

S

T

U

V

W

X