

THE ART & SCIENCE OF JAVASCRIPT

BY CAMERON ADAMS
JAMES EDWARDS
CHRISTIAN HEILMANN
MICHAEL MAHEMOFF
ARA PEHLIVANIAN
DAN WEBB
SIMON WILLISON



INSPIRATIONAL, CUTTING-EDGE JAVASCRIPT FROM THE WORLD'S BEST

The Art & Science of JavaScript (Sample Chapters)

Thank you for downloading these sample chapters of *The Art & Science of JavaScript*, published by SitePoint.

This excerpt includes the Summary of Contents, Information about the Author, Editors and SitePoint, Table of Contents, Preface, two chapters from the book, and the index.

We hope you find this information useful in evaluating this book.

[For more information or to order, visit sitepoint.com](http://sitepoint.com)

Summary of Contents of this Excerpt

	Preface	xiii
CHAPTER 1	Fun With Tables	1
CHAPTER 6	Building a 3D Maze with CSS and JavaScript	189
	Index	251

Summary of Additional Book Contents

CHAPTER 2	Creating Client-side Badges	45
CHAPTER 3	Vector Graphics with canvas	75
CHAPTER 4	Debugging and Profiling with Firebug	121
CHAPTER 5	Metaprogramming with JavaScript	149
CHAPTER 7	Flickr and Google Maps Mashups	217

The Art & Science Of JavaScript

by Cameron Adams, James Edwards, Christian Heilmann, Michael Mahemoff, Ara Pehlivanian, Dan Webb, and Simon Willison

Copyright © 2007 SitePoint Pty. Ltd.

Expert Reviewer: Robert Otani

Managing Editor: Simon Mackie

Technical Editor: Matthew Magain

Technical Director: Kevin Yank

Printing History:

First Edition: January 2008

Editor: Georgina Laidlaw

Index Editor: Fred Brown

Cover Design: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.

Reprint Permissions

To license parts of this book for photocopying, email distribution, intranet or extranet posting, or for inclusion in a course pack, visit <http://www.copyright.com>, and enter this book's title or ISBN to purchase a reproduction license.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood

VIC Australia 3066.

Web: www.sitepoint.com

Email: business@sitepoint.com

ISBN 978-0-9802858-4-0

Printed and bound in Canada

About the Authors

Cameron Adams—The Man in Blue¹—melds a background in computer science with over eight years' experience in graphic design to create a unique approach to interface design. Using the latest technologies, he likes to play in the intersection between design and code to produce innovative but usable sites and applications. In addition to the projects he's currently tinkering with, Cameron has taught numerous workshops and spoken at conferences worldwide, including @media, Web Directions, and South by South West. Every now and then he likes to sneak into bookshops and take pictures of his own books, which have been written on topics ranging from JavaScript to CSS and design. His latest publication, *Simply JavaScript*, takes a bottom-up, quirky-down approach to the basics of JavaScript coding.

James Edwards says of himself:

In spring, writes, and builds
Standards and access matters
Hopes for sun, and rain

Chris Heilmann has been a web developer for ten years, after dabbling in radio journalism. He works for Yahoo in the UK as trainer and lead developer, and oversees the code quality on the front end for Europe and Asia. He blogs at <http://wait-till-i.com> and is available on many a social network as “codepo8.”²

Michael Mahemoff³ is a hands-on software architect with 23 years of programming experience, 12 years commercially. Building on psychology and software engineering degrees, he completed a PhD in design patterns for usability at the University of Melbourne.⁴ He documented 70 Ajax patterns—spanning technical design, usability, and debugging techniques—in the aptly-named *Ajax Design Patterns* (published by O'Reilly) and is the founder of the popular AjaxPatterns.org wiki. Michael is a recovering Java developer, with his programming efforts these days based mostly on Ruby/Rails, PHP and, of course, JavaScript. Lots of JavaScript. You can look up his blog and podcast, where he covers Ajax, software development, and usability, at <http://softwareas.com/>.

Ara Pehlivanian has been working on the Web since 1997. He's been a freelancer, a webmaster, and most recently, a front-end architect and team lead for Nurun, a global interactive communications agency. Ara's experience comes from having worked on every aspect of web development throughout his career, but he's now following his passion for web standards-based front-end development. When he isn't teaching about best practices or writing code professionally, he's maintaining his personal site at <http://arapehlivanian.com/>.

Dan Webb is a freelance web application developer whose recent work includes developing Event Wax, a web-based event management system, and Fridaycities, a thriving community site for Londoners. He maintains several open source projects including Low Pro and its predecessor, the Unobtrusive JavaScript Plugin for Rails, and is also a member of the Prototype core team. He's been a JavaScript programmer for seven years and has spoken at previous @media conferences, RailsConf, and The Ajax Experience. He's also written for A List Apart, HTML Dog, SitePoint and *.NET Magazine*. He blogs regularly about Ruby, Rails and JavaScript at his site, danwebb.net, and wastes all his cash on hip hop records and rare sneakers.

¹ <http://www.themaninblue.com>

² Christian Heilmann photo credit: Philip Tellis [<http://www.flickr.com/photos/bluesmoon/1545636474/>]

³ <http://mahemoff.com/>

⁴ <http://mahemoff.com/paper/patternLanguages.shtml>

Simon Willison is a seasoned web developer from the UK. He is the co-creator of the Django web framework⁵ and a long-time proponent of unobtrusive scripting.

About the Expert Reviewer

Robert Otani enjoys working with brilliant people who make products that enhance the way people think, see, and communicate. While pursuing a graduate degree in physics, Robert caught onto web development as a career, starting with game developer Psygnosis, and has held software design and engineering positions at Vitria, AvantGo, and Sybase. He is currently working with the very talented crew at IMVU,⁶ where people can express their creativity and socialize by building their own virtual worlds in 3D, and on the Web. He enjoys his time away from the keyboard with his wife Alicia and their two dogs, Zeus and Stella. His personal web site can be found at <http://www.otanistudio.com>.

About the Technical Editor

Before joining the SitePoint team as a technical editor, Matthew Magain worked as a software developer for IBM and also spent several years teaching English in Japan. He is the organizer for Melbourne's Web Standards Group,⁷ and enjoys candlelit dinners and long walks on the beach. He also enjoys writing bios that sound like they belong in the personals column. Matthew lives with his wife Kimberley and daughter Sophia.

About the Technical Director

As Technical Director for SitePoint, Kevin Yank oversees all of its technical publications—books, articles, newsletters, and blogs. He has written over 50 articles for SitePoint, but is best known for his book, *Build Your Own Database Driven Website Using PHP & MySQL*. Kevin lives in Melbourne, Australia, and enjoys performing improvised comedy theater and flying light aircraft.

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our books, newsletters, articles, and community forums.

⁵ <http://www.djangoproject.com/>

⁶ <http://www.imvu.com>

⁷ <http://webstandardsgroup.org/>

Table of Contents

Preface	xiii
Who Should Read This Book?	xiii
What's Covered in This Book?	xiv
The Book's Web Site	xv
The Code Archive	xv
Updates and Errata	xv
The SitePoint Forums	xv
The SitePoint Newsletters	xv
Your Feedback	xvi
Conventions Used in This Book	xvi
Code Samples	xvi
Tips, Notes, and Warnings	xvii
 Chapter 1 Fun with Tables	 1
Anatomy of a Table	1
Accessing Table Elements with getElementById	4
Accessing Table Elements with getElementsByTagName	6
Sortable Columns	7
Making Our Tables Sortable	7
Performing the Sort	12
Creating Draggable Columns	24
Making the Table's Columns Draggable	25
Dragging Columns without a Mouse	37
Summary	44
 Chapter 2 Creating Client-side Badges	 45
Badges—an Introduction	46
Too Many Badges Spoil the Broth	46
Out-of-the-box Badges	48
Server-side Badges	50

Custom Client-side Badges	51
Client-side Badge Options: Ajax and JSON	53
The Problem with Ajax	53
JSON: the Lightweight Native Data Format	54
Providing a Fallback for Failed Connections	58
Planning the Badge Script	59
The Complete Badge Script	61
Defining Configuration Variables	63
Defining Public Methods	64
Defining Private Methods	67
Calling for Server Backup	72
Summary	73
 Chapter 3 Vector Graphics with canvas	75
Working with canvas	76
The canvas API	77
Thinking About Vector Graphics	78
Creating Shapes	79
Creating a Pie Chart	98
Drawing the Chart	98
Casting a Shadow	104
Updating the Chart Dynamically	109
canvas in Internet Explorer	115
Summary	119
 Chapter 4 Debugging and Profiling with Firebug	121
Installing and Running Firebug	122
Installing Firefox and Firebug	122
First Steps with Firebug	123
Opening, Closing, and Resizing Firebug	124
Enabling and Disabling Firebug	127
The Many Faces of Firebug	127
Common Components	127

The Firebug Views	128
Switching Views	132
Using Firebug	133
Performing Rapid Application Development	133
Monitoring, Logging, and Executing with the Console	134
Viewing and Editing On the Fly	138
Debugging Your Application	140
Performance Tuning Your Application	143
Related Tools	145
Firebug Lite	145
YSlow	146
Microsoft Tools	146
Other Firefox Extensions	147
Summary	147
 Chapter 5 Metaprogramming with JavaScript	149
The Building Blocks	150
(Nearly) Everything Is a Hash	150
Finding and Iterating through Properties in an Object	151
Detecting Types	152
There Are No Classes in JavaScript	153
Detecting whether a Function Was Called with new	154
Functions Are Objects	155
Understanding the arguments Array	157
Comprehending Closures	159
Metaprogramming Techniques	164
Creating Functions with Default Arguments	164
Working with Built-ins	165
Creating Self-optimizing Functions	168
Aspect-oriented Programming on a Shoestring	171
Better APIs through Dynamic Functions	172
Creating Dynamic Constructors	176
Simulating Traditional Object Orientation	178

Summary	187
---------------	-----

Chapter 6 Building a 3D Maze with CSS and JavaScript

Basic Principles	190
Making Triangles	191
Defining the Floor Plan	193
Creating Perspective	196
Making a Dynamic View	198
Core Methods	198
Applying the Finishing Touches	208
Limitations of This Approach	209
Creating the Map View	209
Adding Captions	212
Designing a Floor Plan	213
Further Developments	214
Using the Callback	214
Blue-sky Possibilities	215
Summary	216

Chapter 7 Flickr and Google Maps Mashups

APIs, Mashups, and Widgets! Oh, My!	218
Flickr and Google Maps	218
Drawing a Map	219
Geotagging Photos	221
Getting at the Data	222
JSON	223
The Same-origin Restriction	224
Pulling it All Together	233
Enhancing Our Widget	238
Putting it All Together	245
Taking Things Further	249
Summary	250

Index 251

Preface

Once upon a time, JavaScript was a dirty word.

It got its bad name from being misused and abused—in the early days of the Web, developers only ever used JavaScript to create annoying animations or unnecessary, flashy distractions.

Thankfully, those days are well behind us, and this book will show you just how far we’ve come. It reflects something of a turning point in JavaScript development—many of the effects and techniques described in these pages were thought impossible only a few years ago.

Because it has matured as a language, JavaScript has become enormously trendy, and a plethora of frameworks have evolved around many of the best practice techniques that have emerged with renewed interest in the language. As long-time JavaScript enthusiasts, we’ve always known that the language had huge potential, and nowadays, much of the polish that makes a modern web application really stand out is usually implemented with JavaScript. If CSS was the darling of the early 2000s, JavaScript has since well and truly taken over the throne.

In this book, we’ve assembled a team of experts in their field—a veritable who’s who of JavaScript developers—to help you take your JavaScript skills to the next level. From creating impressive mashups and stunning, dynamic graphics to more subtle user-experience enhancements, you’re about to open Pandora’s box. At a bare minimum, once you’ve seen what’s possible with the new JavaScript, you’ll likely use the code in this book to create amazing user experiences for your users. Of course, if you have the inclination, you may well use your new-found knowledge to change the world.

We look forward to buying a round of drinks at your site’s launch party!

Who Should Read This Book?

This book is targeted at intermediate JavaScript developers who want to take their JavaScript skills to the next level without sacrificing web accessibility or best practice. If you’ve never written a line of JavaScript before, this probably isn’t the right book for you—some of the logic in the later chapters can get a little hairy.

If you have only a small amount of experience with JavaScript, but are comfortable enough programming in another language such as PHP or Java, you’ll be just fine—we’ll hold your hand along the way, and all of the code is available for you to download and experiment with on your own. And if you’re an experienced JavaScript developer, we would be very, *very* surprised if you didn’t learn a thing or two. In fact, if you *only* learn a thing or two, you should contact us here at SitePoint—we may have a book project for you to tackle!

What's Covered in This Book?

Chapter 1: Fun with Tables

HTML tables get a bad rap among web developers, either because of their years of misuse in page layouts, or because they can be just plain boring. In this chapter, Ara Pehlivanian sets out to prove that not only are properly used tables *not* boring, but they can, in fact, be a lot of fun—especially when they're combined with some JavaScript. He introduces you to the DOM, then shows how to make table columns sortable and draggable with either the mouse or the keyboard.

Chapter 2: Creating Client-side Badges

Badges are snippets of third-party data (image thumbnails, links, and so on) that you can add to your blog to give it some extra personality. Christian Heilmann walks us through the task of creating one for your own site from scratch, using JSON and allowing for a plan B if the connection to the third-party server dies.

Chapter 3: Creating Vector Graphics with canvas

In this chapter, Cameron Adams introduces the canvas element, and shows how you can use it to create vector graphics—from static illustrations, to database driven graphs and pie charts—that work across all modern browsers. After you've read this chapter, you'll never look at graphics on the Web the same way again!

Chapter 4: Debugging and Profiling with Firebug

Firebug is a plugin for the Firefox browser, but calling it a plugin doesn't do it justice—Firebug is a full-blown editing, debugging, and profiling tool. It takes the traditionally awkward task of JavaScript debugging and optimization, and makes it intuitive and fun. Here, Michael Mahemoff reveals tons of pro-level tips and hidden treasures to give you new insight into this indispensable development tool.

Chapter 5: Metaprogramming with JavaScript

Here, Dan Webb takes us on a journey into the mechanics of the JavaScript language. By understanding a little about the theory of metaprogramming, he shows how we can use JavaScript to extend the language itself, improving its object oriented capabilities, improving support for older browsers, and adding methods and operators that make JavaScript development more convenient.

Chapter 6: Building a 3D Maze with CSS and JavaScript

Just when you thought you'd seen everything, James Edwards shows you how to push the technologies of CSS and JavaScript to their limits, as he creates a real game in which the player must navigate around a 3D maze! Complete with a floor-plan generator and accessibility features like keyboard navigation and captions, this chapter highlights the fact that JavaScript's potential is limited only by one's imagination.

Chapter 7: Flickr and Google Maps Mashups

Ever wished you could combine the Web's best photo-management site, Flickr, with the Web's best mapping service, Google Maps, to create your own über-application? Well, you can! Simon Willison shows that, by utilizing the power of JavaScript APIs, creating a mashup from two third-party web sites is easier than you might have thought.

The Book's Web Site

Located at <http://www.sitepoint.com/books/jsdesign1/>, the web site that supports this book will give you access to the following facilities.

The Code Archive

As you progress through this book, you'll note file names above many of the code listings. These refer to files in the code archive—a downloadable ZIP file that contains all of the finished examples presented in this book. Simply click the **Code Archive** link on the book's web site to download it.

Updates and Errata

No book is error-free, and attentive readers will no doubt spot at least one or two mistakes in this one. The Corrections and Typos page on the book's web site will provide the latest information about known typographical and code errors, and will offer necessary updates for new releases of browsers and related standards.¹

The SitePoint Forums

If you'd like to communicate with other web developers about this book, you should join SitePoint's online community.² The JavaScript forum,³ in particular, offers an abundance of information above and beyond the solutions in this book, and a lot of fun and experienced JavaScript developers hang out there. It's a good way to learn new tricks, get questions answered in a hurry, and just have a good time.

The SitePoint Newsletters

In addition to books like this one, SitePoint publishes free email newsletters including *The SitePoint Tribune*, *The SitePoint Tech Times*, and *The SitePoint Design View*. Reading them will keep you up to date on the latest news, product releases, trends, tips, and techniques for all aspects of web development. Sign up to one or more SitePoint newsletters at <http://www.sitepoint.com/newsletter/>.

¹ <http://www.sitepoint.com/books/jsdesign1/errata.php>

² <http://www.sitepoint.com/forums/>

³ <http://www.sitepoint.com/launch/jsforum/>

Your Feedback

If you can't find an answer through the forums, or if you wish to contact us for any other reason, the best place to write is books@sitepoint.com. We have an email support system set up to track your inquiries, and friendly support staff members who can answer your questions. Suggestions for improvements as well as notices of any mistakes you may find are especially welcome.

Conventions Used in This Book

You'll notice that we've used certain typographic and layout styles throughout this book to signify different types of information. Look out for the following items.

Code Samples

Code in this book will be displayed using a fixed-width font, like so:

```
<h1>A perfect summer's day</h1> <p>It
    was a lovely day for a walk in the park. The birds were
    singing and the kids were all back at school.</p>
```

If the code may be found in the book's code archive, the name of the file will appear at the top of the program listing, like this:

example.css

```
.footer { background-color: #CCC; border-top: 1px
    solid #333; }
```

If only part of the file is displayed, this is indicated by the word *excerpt*:

example.css (*excerpt*)

```
border-top: 1px solid #333;
```

Some lines of code are intended to be entered on one line, but we've had to wrap them because of page constraints. A ➤ indicates a page-break that exists for formatting purposes only, and should be ignored. A vertical ellipsis (:) refers to code that has been omitted from the example listing to conserve space.

```
if (a == b) {
    :
}
URL.open("http://www.sitepoint.com/blogs/2007/11/01/the-php-anthology-101-essent
➤ial-tips-tricks-hacks-2nd-edition");
```


Tips, Notes, and Warnings



Hey, You!

Tips will give you helpful little pointers.



Ahem, Excuse Me ...

Notes are useful asides that are related—but not critical—to the topic at hand. Think of them as extra tidbits of information.



Make Sure You Always ...

... pay attention to these important points.



Watch Out!

Warnings will highlight any gotchas that are likely to trip you up along the way.

Chapter 1

Fun with Tables

For the longest time, tables were the tool of choice for web designers who needed a non-linear way to lay out a web page's contents. While they were never intended to be used for this purpose, the row-and-column structure of tables provided a natural grid system that was too tempting for designers to ignore. This misuse of tables has shifted many designers' attention away from the original purpose for which they were intended: the marking up of tabular data.

Though the life of a table begins in HTML, it doesn't have to end there. JavaScript allows us to add interactivity to an otherwise static HTML table. The aim of this chapter is to give you a solid understanding of how to work with tables in JavaScript, so that once you've got a grip on the fundamentals, you'll be comfortable enough to go well beyond the examples provided here, to do some wild and crazy things of your own.

If you're new to working with the DOM, you'll also find that this chapter doubles as a good introduction to DOM manipulation techniques, which I'll explain in as much detail as possible.

Anatomy of a Table

Before we can have fun with tables, it's important to cover some of the basics. Once we have a good understanding of a table's structure in HTML, we'll be able to manipulate it more easily and effectively with JavaScript.

In the introduction I mentioned a table's row-and-column structure. In fact, there's no such thing as columns in a table—at least, not in an HTML table. The columns are an illusion. Structurally, a table is a collection of rows, which in turn are collections of cells. There is no tangible HTML element

that represents a column of cells—the only elements that come close are `colgroup` and `col`, but they serve only as aids in styling the table. In terms of actual structure, there are no columns.

Let's take a closer look at the simple table shown in Figure 1.1.

Companies	Q1	Q2	Q3	Q4
Company A	\$621	\$942	\$224	\$486
Company B	\$147	\$1,325	\$683	\$524
Company C	\$135	\$2,342	\$33	\$464
Company D	\$164	\$332	\$331	\$438
Company E	\$199	\$902	\$336	\$1,427

*Stated in millions of dollars

Figure 1.1. A simple table

I've styled the table with some CSS in order to make it a little easier on the eyes. The markup looks like this:

simple.html (excerpt)

```
<table id="sales" summary="Quarterly sales figures for competing
companies. The figures are stated in millions of dollars.">
  <caption>Quarterly Sales*</caption>
  <thead>
    <tr>
      <th scope="col">Companies</th>
      <th scope="col">Q1</th>
      <th scope="col">Q2</th>
      <th scope="col">Q3</th>
      <th scope="col">Q4</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <th scope="row">Company A</th>
      <td>$621</td>
```

```

        <td>$942</td>
        <td>$224</td>
        <td>$486</td>
    </tr>
    <tr>
        <th scope="row">Company B</th>
        <td>$147</td>
        <td>$1,325</td>
        <td>$683</td>
        <td>$524</td>
    </tr>
    <tr>
        <th scope="row">Company C</th>
        <td>$135</td>
        <td>$2,342</td>
        <td>$33</td>
        <td>$464</td>
    </tr>
    <tr>
        <th scope="row">Company D</th>
        <td>$164</td>
        <td>$332</td>
        <td>$331</td>
        <td>$438</td>
    </tr>
    <tr>
        <th scope="row">Company E</th>
        <td>$199</td>
        <td>$902</td>
        <td>$336</td>
        <td>$1,427</td>
    </tr>
</tbody>
</table>
<p class="footnote">*Stated in millions of dollars</p>

```

Each set of `<tr></tr>` tags tells the browser to begin a new row in our table. The `<th>` and `<td>` tags inside them represent header and data cells, respectively. Though the cells are arranged vertically in HTML, and almost look like columns of data, they're actually rendered horizontally as part of a row.

Notice also that the rows are grouped within either `<thead>` or a `<tbody>` tags. This not only provides a clearer semantic structure, but it makes life easier when we're working with the table using JavaScript, as we'll see in a moment.

Accessing Table Elements with `getElementById`

When our browser renders a page, it constructs a DOM tree of that page. Once this DOM tree has been created, we're able to access elements of our table using a range of native DOM methods.

The `getElementById` method is one that you'll see in most of the chapters of this book. Here's an example of how we'd use it to access a table's rows:

```
var sales = document.getElementById("sales");  
var rows = sales.rows;
```

In the above code, we first obtain a reference to our table using the DOM method `getElementById`, and place it into a variable named `sales`. We then use this variable to obtain a reference to the collection of table rows. We place this reference into a variable named, quite aptly, `rows`.

The example above is all very well, but what if we only wanted the row inside the `thead` element? Or maybe just the ones located inside the `tbody`? Well, those different groups of rows are also reflected in the DOM tree, and we can access them with the following code:

```
var sales = document.getElementById("sales");  
var headRow = sales.tHead.rows;  
var bodyRows = sales.tBodies[0].rows;
```

As the above code demonstrates, accessing the row inside the `thead` is fairly straightforward. You'll notice that getting at the `tbody` rows is a little different, however, because a table can have more than one `tbody`. What we're doing here is specifying that we want the collection of rows for the first `tbody` in the `tBodies` collection. As collections begin counting at zero, just like arrays, the first item in the collection is actually item 0, and can be accessed using `tBodies[0]`.



Who's This DOM Guy, Anyway?

The **Document Object Model (DOM)** is a standardized API for programmatically working with markup languages such as HTML and XML.

The DOM is basically an object oriented representation of our document. Every element in our HTML document is represented by an object in that document's DOM tree. These objects—referred to as **nodes**—are organized in a structure that mirrors the nested HTML elements that they represent, much like a tree.

The DOM tree also contains objects whose job is to help make working with our document easier; one example is the following code's `rows` object, which doesn't exist in our source HTML document. And each object in the DOM tree contains supplementary information regarding, among other things, its position, contents, and physical dimensions.

We follow the same principle to access a particular row. Let's get our hands on the first row inside the first `tbody`:

```
var sales = document.getElementById("sales");
var bodyRows = sales.tBodies[0].rows;
var row = bodyRows[0];
```

Of course, JavaScript offers us many ways to achieve the same goal. Take a look at this example:

```
var sales = document.getElementById("sales");
var tBody = sales.tBodies[0];
var rows = tBody.rows;
var row = rows[0];
```

The result of that code could also be represented by just one line:

```
var row = document.getElementById("sales").tBodies[0].rows[0];
```

In the end, the approach you choose should strike the right balance between efficiency and legibility. Four lines of code may be considered too verbose for accessing a row, but a one-line execution may be difficult to read. The single line above is also more error prone than the four-row example, as that code doesn't allow us to check for the existence of a collection before accessing its children.

Of course, you could go to the other extreme:

```
var sales = document.getElementById("sales");
if (sales) {
  var tBody = sales.tBodies[0];
  if (tBody) {
    var rows = tBody.rows;
    if (rows) {
      var row = rows[0];
    }
  }
}
```

This code checks your results every step of the way before it proceeds, making it the most robust of the above three code snippets. After all, it's possible that the table you're accessing doesn't contain a `tbody` element—or any rows at all! In general, I favor robustness over terseness—racing towards the first row without checking for the existence of a `tbody`, as we've done in our one-line example, is likely to result in an uncaught error for at least some users. We'll discuss some guidelines for deciding on an appropriate coding strategy in the coming sections.

We follow the same principles to access the cells in a table: each row contains a `cells` collection which, as you might have guessed, contains references to all of the cells in that row. So, to access the first cell in the first row of a table, we can write something like this:

```
var sales = document.getElementById("sales");
var cell = sales.rows[0].cells[0];
```

Here, we've ignored the fact that there may be a `tHead` or a `tBodies` collection, so the row whose cells we're accessing is the first row in the table—which, as it turns out, is the row in the `thead`.

Accessing Table Elements with `getElementsByTagName`

We have at our disposal a number of ways to access table information—we aren't restricted to using collections. For example, you might use the general-purpose DOM method `getElementsByTagName`, which returns the children of a given element. Using it, we can grab all of the `td`s in a table, like this:

```
var sales = document.getElementById("sales");
var tds = sales.getElementsByTagName("td");
```

Those two lines of code make a convenient alternative to this much slower and bulkier option:

```
var sales = document.getElementById("sales");
var tds = [];
for (var i=0; i<sales.rows.length; i++) {
  for (var j=0; j<sales.rows[i].cells.length; j++) {
    if (sales.rows[i].cells[j].nodeName == "TD") {
      tds.push(sales.rows[i].cells[j]);
    }
  }
}
```

Of course, choosing which technique you'll use is a question of using the right tool for the job. One factor you'll need to consider is performance; another is how maintainable and legible your code needs to be. Sometimes, though, the choice isn't obvious. Take a look at the following examples. Here's the first:

```
var sales = document.getElementById("sales");
var cells = sales.rows[1].cells;
```

And here's the second:

```
var sales = document.getElementById("sales");
var cells = sales.rows[1].getElementsByTagName("*");
```


Both of these code snippets produce the same results. Neither uses any `for` loops; they're both two lines long; and they both reach the `cells` collection through `sales.rows[1]`. But one references the `cells` collection directly, while the other uses `getElementsByName`.

Now if speed was our main concern, the first technique would be the right choice. Why? Well, because `getElementsByName` is a generic function that needs to crawl the DOM to fetch our cells. The `cells` collection, on the other hand, is specifically tailored for the task.

However, if flexibility was our main concern (for example, if you only wanted to access the `td` elements, not the surrounding elements that form part of the table's hierarchy), `getElementsByName` would be much more convenient. Otherwise, we'd need to loop over the `cells` collection to filter out all of the `th` elements it returned along with the `td` elements.

Sortable Columns

Now that we know how to work with our table through the DOM, let's add to it some column sorting functionality that's similar to what you might find in a spreadsheet application. We'll implement this feature so that clicking on a column's heading will cause its contents to be sorted in either ascending or descending order. We'll also make this behavior as accessible as possible by ensuring that it works with and without a mouse. Additionally, instead of limiting the functionality to one specific table, we'll implement it so that it works with as many of a page's tables as we like. Finally, we'll make our code as easy to add to a page as possible—we'll be able to apply the sorting functionality to any table on the page by including a single line of code.

Making Our Tables Sortable

First, in order to apply our sort code to as many tables as we want, we need to write a function that we can instantiate for each table that we want to make sortable:

tablesort.js (excerpt)

```
function TableSort(id) {
  this.tbl = document.getElementById(id);
  if (this.tbl && this.tbl.nodeName == "TABLE") {
    this.makeSortable();
  }
}
```

In the code above, we've created a function named `TableSort`. When it's called, `TableSort` takes the value in the `id` parameter, and fetches a reference to an element on the page using the `getElementById` DOM method. We store this reference in the variable `this.tbl`. If we find a valid element with that `id`, and that element is a table, we can make it sortable by calling the `makeSortable` function.

Making the Sort Functionality Accessible

I mentioned that we'd make our sorting functionality accessible to both mouse and keyboard users. Here's the code that will help us achieve this:

tablesort.js (excerpt)

```
TableSort.prototype.makeSortable = function () { ❶
  var headings = this.tbl.tHead.rows[0].cells;
  for (var i=0; headings[i]; i++) { ❷
    headings[i].cIdx = i; ❸
    var a = document.createElement("a");
    a.href = "#";
    a.innerHTML = headings[i].innerHTML; ❹
    a.onclick = function (that) { ❺
      return function () {
        that.sortCol(this);
        return false;
      }
    }(this);
    headings[i].innerHTML = ""; ❻
    headings[i].appendChild(a);
  }
}
```

We wrap anchors (a elements) around the contents of each of the th elements. This enables our users to tab to an anchor, and to trigger an onclick event by pressing the **Enter** key—thereby allowing the sort to be activated without a mouse. And, since we're adding these anchors dynamically, we don't need to worry about them cluttering up our markup.

Here are the details:

- ❶ We're assigning our function to the object's prototype object. The prototype object simplifies the process of adding custom properties or methods to all instances of an object. It's a powerful feature of JavaScript that's used heavily throughout the rest of this book, particularly in the field of metaprogramming (see Chapter 5).
- ❷ We iterate over the cells in the headings collection with a for loop. This loop is slightly different from those we've seen in previous examples in that the condition that's checked on each pass is headings[i], rather than the traditional i < headings.length. This is an optimization technique: the for loop checks to see if there's an item in the headings collection at position i, and avoids having to calculate the length of the array on each pass.

Avoiding length can save valuable milliseconds if you're dealing with large datasets, though in our case—with only five items in our array—this approach is just shorter (and quicker to write).

- 3 Because of a bug in Safari 2.0.4 that causes the browser to always return a value of 0 for `cellIndex`, we need to emulate the `cellIndex` value for each heading. We do so by assigning the value of `i` to a new property that we've created, called `cIdx`.
- 4 Inside the loop, we create a new anchor, and copy the contents of the `th` into it.
- 5 We also add an `onclick` event to the anchor, the reason for which I'll explain in a moment.
- 6 Finally, we clear the original contents of the `th` and insert the new anchor in place of the original contents.



Using `innerHTML`

I'm using the `innerHTML` property here; even though it isn't part of the W3C recommendation, it's widely supported and operates faster (as well as being much simpler to use) than the myriad DOM methods I'd have to use to achieve the same outcome.



Making Assumptions is Okay ... Sometimes

You'll note that we're grabbing the `th` cells in the table's `thead` with only one line of code, taking for granted that a `thead` exists, and that it contains at least one row. I've done so for brevity—in this example, we can be certain of the contents of the table that we're working with. If we didn't have the same control over the table on which we were operating, we'd have to use more verbose code to check each step along the way, as demonstrated earlier.

Handling Events and Scope Issues

You'll notice that we assigned an `onclick` event to the anchor just before we added it to the page.

The reason why we've used nested functions here is to fix a scope problem with the `this` keyword. In a function that's called by an event such as `onclick`, the `this` keyword refers to the calling element. However, in this case, we need `this` to point to our instance of `TableSort`—not the anchor that was just clicked—so that our `onclick` code can access the `this.tbl` property.

Normally, we'd assign an anonymous function directly to our anchor's `onclick` event handler like this:

```
a.onclick = function () {
    alert(this);
}
```

But if we were to take this approach, the `this` keyword would return a reference to the anchor element that was just clicked. Instead, we replace it with a self-executing function that returns another

anonymous function to the `onclick` handler. The outer function forms a **closure**, a concept that's discussed in more detail in Chapter 5. Here's the revised code:

```
a.onclick = function (that) {
  return function () {
  }
}(this);
```

The brackets at the end of the main function cause it to be executed as soon as it is loaded by the browser (rather than when the click event is triggered); the parameter passed to the function is the current `this` keyword, which refers to our `TableSort` instance.

Inside the function, we receive the `TableSort` reference in a variable named `that`. Then, in the scope of the actual `onclick` event, all we have to do is call our `sortCol` function and pass *it* the reference to the anchor element, like so:

```
a.onclick = function (that) {
  return function () {
    that.sortCol(this);
    return false;
  }
}(this);
```

Here, `return false;` ensures that the anchor doesn't try to follow its `href` value—the default behavior for an anchor element. The use of `return false;` is important here, because if our `href` value was `#`, it would add the click to our browser's history, and return users to the top of the page if they weren't there already.

Adding Some Class

Now that we've wrapped our `th` elements with anchors and added a few more lines of CSS, we have a table that looks like the one in Figure 1.2.



Be Careful when Assigning Events

In this example, I've assigned functions directly to our event handlers as a way to minimize the size of the script. However, this can be a dangerous practice! Our use of code like `a.onclick = function () { ... }` creates a one-to-one relationship between our event and the function. So if, previously, a function had been assigned to the event handler, this code would overwrite it. To make your scripts more robust, consider using one of the many `addEventListener` functions available online. I'm a big fan of the Yahoo! User Interface Library's `addListener` function.¹

¹ <http://developer.yahoo.com/yui/event/>

QUARTERLY SALES*

Companies ▲	Q1 ▲	Q2 ▲	Q3 ▲	Q4 ▲
Company A	\$621	\$942	\$224	\$486
Company B	\$147	\$1,325	\$683	\$524
Company C	\$135	\$2,342	\$33	\$464
Company D	\$164	\$332	\$331	\$438
Company E	\$199	\$902	\$336	\$1,427

*Stated in millions of dollars

Figure 1.2. A table ready to be sorted

The arrows next to the column headings signify that each column can be used as a basis to sort the table's data. The active column's arrow is darkened; the inactive columns' arrows are dimmed to show that, though they're clickable, they aren't currently being used to sort the table's content. The arrows are inserted as CSS background images on the newly inserted `a` elements, and are managed with two class names: `asc` for ascending and `dsc` for descending. Heading cells without an `asc` or `dsc` class name receive an inactive arrow.

These class names aren't included just for decorative purposes. We'll be using them in our code to identify the direction in which a column is being sorted, and to toggle the sort direction when a user clicks on a heading. Here's how our table begins:

tablesort.html (excerpt)

```
<thead>
  <tr>
    <th class="asc" scope="col">Companies</th>
    <th scope="col">Q1</th>
    <th scope="col">Q2</th>
    <th scope="col">Q3</th>
    <th scope="col">Q4</th>
  </tr>
</thead>
```

In this code, we've added the class name `asc` to the first column heading because we know that the companies are sorted alphabetically in the markup.

Performing the Sort

Once the users click on a heading (technically speaking, they click on the anchor surrounding the heading), the `sortCol` function is called, and is passed a reference to the calling (clicked) element. This process is important, as it identifies to us the column that we need to sort.

Our first order of business is to set up a few important variables:

tablesort.js (excerpt)

```
TableSort.prototype.sortCol = function (e1) {
  var rows = this.tbl.rows;
  var alpha = [], numeric = [];
  var aIdx = 0, nIdx = 0;
  var th = e1.parentNode;
  var cellIndex = th.cIdx;
  :
}
```

Here's a description of each of these variables:

rows	This variable is a shortcut to the table's rows; it lets us avoid having to type <code>this.tbl.rows</code> each time we want to refer to them.
alpha, numeric	These arrays will allow us to store the alphanumeric and numeric contents of the cells in our column.
aIdx, nIdx	These two variables are indices to be used with our two arrays. We'll increment them individually every time we add an item to one of the arrays.
th	This is a reference to the clicked anchor's parent, which is a <code>th</code> . The anchor's reference is <code>e1</code> , which is passed as a parameter to the <code>sortCol</code> function.
cellIndex	This variable stores the <code>th</code> element's index within its parent row. Using the <code>cellIndex</code> value, we'll be able to skip to the correct cell in each row, effectively traversing a column of cells.

Parsing the Content

Now let's loop over the table's rows and process each cell that falls beneath the heading that was clicked. The first thing we're going to have to do is retrieve the cell's contents—regardless of whether that data is found inside nested `` or `` tags, for example:

tablesort.js (excerpt)

```
for (var i=1; rows[i]; i++) {
  var cell = rows[i].cells[cellIndex];
  var content =
    cell.textContent ? cell.textContent : cell.innerText;
  :
```

We're after the final data value of the cell, regardless of whether that value is simply \$150, `$150`, or `<p>$150</p>`. The simplest way to achieve this goal is to use either the `textContent` or `innerText` properties of the element. Firefox supports only `textContent`, and Internet Explorer supports only `innerText`, while Safari and Opera support both. So in order to make the code work across the board, we'll use a ternary operator that uses one property or the other, depending on what's available in the browser executing the script.

Now that we've grabbed the cell's contents, we need to determine whether the data we retrieved is alphanumeric or numeric. We need to do this because we need to sort the two types of data separately—alphanumeric data should come *after* numeric data in our output.²



Using the Ternary Operator to Normalize Browser Inconsistencies

Sometimes you'll encounter situations where browser makers have decided not to follow the W3C spec, or for whatever reason, have implement proprietary functionality. Either way, the resulting inconsistencies can be a headache when you're trying to support multiple browsers.

Using a traditional `if/else` statement can be a bit bulky when all you want is to assign a value to your variable from two potentially different sources, depending on which is available. Enter: the **ternary operator**, a compressed `if/else` statement with the syntax `condition ? true : false`; Let's consider its application in terms of this code:

```
if (cell.textContent) {
  var content = cell.textContent;
} else {
  var content = cell.innerText ;
}
```

We can use the ternary operator to replace the above with this single line of code (well, it would be a single line if we could fit it on this page!):

```
var content = cell.textContent ?
  cell.textContent : cell.innerText;
```

² If we were to expand our table sort to accommodate other types of data—such as dates—we'd need to separate our data even further. For this demo, however, we'll restrict our sort functionality to include only alphabetic and numeric data.

Here's the code:

tablesort.js (excerpt)

```
var num = content.replace(/(\$|\\,|\\s)/g, ""); ❶
if (parseFloat(num) == num) { ❷
    numeric[nIdx++] = {
        value: Number(num),
        row: rows[i]
    }
} else {
    alpha[aIdx++] = { ❸
        value: content,
        row: rows[i]
    }
}
```

- ❶ Before we check the data type, we need to strip the cell's contents of any characters that could be used in a numerical context—dollar signs, commas, spaces, and so on—but which might cause numeric data to be interpreted as alphanumeric data. We use a regular expression and the `replace` method to achieve this result.
- ❷ Once we've stripped out the characters, we use JavaScript's `parseFloat` function to see whether or not the remaining cell value is a number. If it is, we store the stripped-down version of it in the `numeric` array.
- ❸ If it isn't a number, we store the untouched cell value in the `alpha` array.

You'll note that we're storing the value in an object literal, which allows us also to store a reference to the row in which the cell was originally found. This row reference will be crucial later, when we reorder the table according to our sort outcome.

Implementing a Bubble Sort

Now that our column's contents are parsed and ready to be sorted, let's take a look at our sort algorithm. If we wanted to, we could use JavaScript's built in `sort` method, which looks like this:

```
var arr = [19, 2, 77, 111, 33, 8];
var sorted = arr.sort();
```

This approach would produce the array `[111, 19, 2, 33, 77, 8]`, which isn't good enough, since the items have been sorted as if they were strings. Luckily, the `sort` method allows us to pass it a comparison function. This function accepts two parameters, and returns either a negative number, a positive number, or zero. If the returned value is negative, it means that the first parameter is

smaller than the second. If the returned value is a positive number, the first parameter is greater than the second. And if the returned value is zero, they're both the same.

Here's how we'd write such a comparison function:

```
function compare(a, b) {
  return a-b;
}
var sorted = unsorted.sort(compare);
```

The trouble with `sort` is that the algorithm itself (that is, the logic it uses to loop over the data) is hard-coded into the browser—the comparison part of the function is the only part you can tweak. Also, `sort` was introduced in JavaScript 1.1—it's not available in older browsers. Of course, supporting outdated browsers isn't a major issue, but if you're a control freak like me, you might want to write your own sort algorithm from scratch, so that it does support older browsers. Let me walk you through an example that shows how to do just this.

We don't need to reinvent the wheel here; we have many different kinds of sort algorithms to choose from—every possible type seems already to have been worked out.³ Even though it's not the quickest algorithm, I've chosen to use a bubble sort⁴ here because it's simple to understand and describe.

A bubble sort works by executing two loops—an inner loop and an outer loop. The inner loop goes over our dataset once and checks the current item against the next item. If the current item is bigger than the next one, it swaps them; if it isn't, the loop moves on to the next item. The outer loop keeps running the inner loop until there are no more items to swap. Figure 1.3 illustrates our table's Q3 data being bubble sorted.

Each iteration of the outer loop is labeled as a “pass,” and each column of data within a pass represents one step forward in the inner loop. The black arrow next to the numbers shows the progress of the inner loop (as does the “i=0” on top of each column). The red, curved arrows represent a swap that has taken place between the current item and the next one. Note how the outer loop goes over the entire dataset once more at the end, to make sure that there aren't any more swaps to perform.

We'd like our function to be able to perform both ascending and descending sorts. Since we don't want to write the same function twice—the two sort algorithms would perform the same operations, only in reverse—we'll write just one `bubbleSort` function, and have it accept a direction parameter as well as the array to be sorted.

³ http://en.wikipedia.org/wiki/Sorting_algorithm

⁴ http://en.wikipedia.org/wiki/Bubble_sort

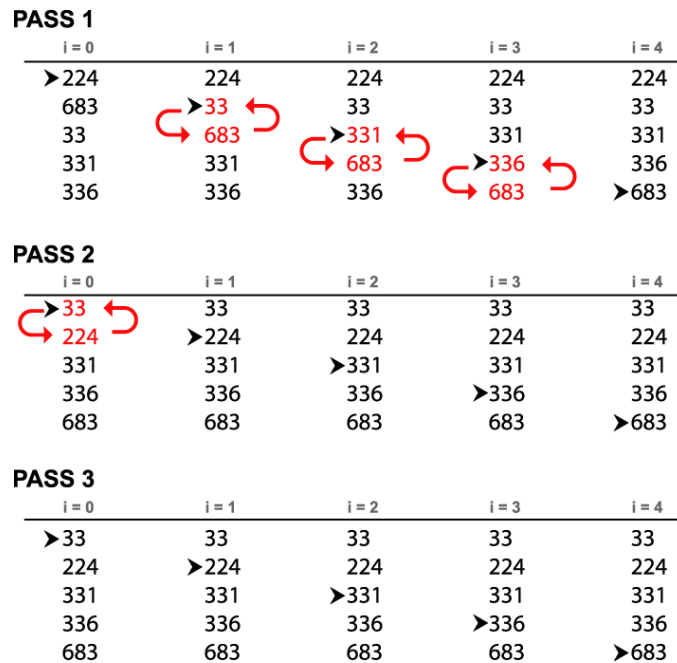


Figure 1.3. A bubble sort in action

There are a couple of things to note here. First, `bubbleSort` is a standalone function and doesn't need to be added to `TableSort` with a prototype object—after all, there's no need to make copies of the function every time a new `TableSort` instance is made. Second, since we'll be using the `dir` parameter to make `bubbleSort` bidirectional, this code may be a little harder to follow than the code we've looked at so far.

Take a deep breath, and let's dive in:

tablesort.js (excerpt)

```
function bubbleSort(arr, dir) {
  var start, end;
  if (dir === 1) { ❶
    start = 0;
    end = arr.length;
  } else if (dir === -1) {
    start = arr.length-1;
    end = -1;
  }
  var unsorted = true;
  while (unsorted) { ❷
    unsorted = false; ❸
    for (var i=start; i!=end; i=i+dir) { ❹
      if (arr[i+dir] && arr[i].value > arr[i+dir].value) { ❺
        var a = arr[i];
        var b = arr[i+dir];
```

```

        var c = a;
        arr[i] = b;
        arr[i+dir] = c;
        unsorted = true;
    }
}
}
return arr;
}

```

Let's take a look at what's going on here:

- ❶ Before we start looping over our data, we need to set up a couple of variables. The `dir` parameter's only valid values, 1 and -1, represent ascending and descending order respectively. Checking for the value of `dir`, we set the start and end points for our inner loop accordingly. When `dir` is ascending, we'll start at zero and end at the array's length; when it's descending, we'll start at the array's length minus one, and end at negative one.
- ❷ I've used a `while` loop for our outer loop, and set it to continue executing until the value for `unsorted` is equal to `false`.
- ❸ For each iteration, we immediately set `unsorted` to `false`, and only set it to `true` in the inner loop if a sort needs to be made.
- ❹ I've used a `for` loop for our inner loop. A `for` loop has three parts to it: an **initialization**, a **condition**, and an **update**:

```

for (initialization; condition; update) {
    // do something
}

```

Our loop criterion looks like this:

```

for (var i=start; i!=end; i=i+dir) {

```

The initialization is set to our `start` value. Our condition returns `true` as long as the counter `i` is not equal to our end value, and we updated our counter by adding the value of `dir` to it.

In the case of an ascending loop, our counter is initialized to zero. The loop continues executing as long as the counter's value does not equal the array's length; the counter is incremented by one with each iteration.

In a descending loop, the counter is initialized to a value that's equal to the array's length minus one, since the array's index counts from zero. The counter is decremented by a value

of one on each iteration, and the loop continues until the counter's value equals -1. This might seem confusing, but this criterion works because `dir` is a negative value, so 1 is subtracted from our counter on each pass.

- 5 Now that we've got our loop working in both directions, we need to check whether the next item in the list is larger than the current one. If it is, we'll swap them:

```
if (arr[i+dir] && arr[i].value > arr[i+dir].value) {
  var a = arr[i];
  var b = arr[i+dir];
  var c = a;
  arr[i] = b;
  arr[i+dir] = c;
  unsorted = true;
}
```

Our `if` statement is made up of two parts. First, to make sure that there is a neighboring item to check against, we try to access `arr[i+dir]`. Since `dir` can be a negative or positive number, this statement will check the item either before or after the current item in the array. If there's an item in that position, our attempt will return `true`. This will allow us to check whether the value of the current item is greater than that of its neighbor. If it is, we need to swap the two.

We also set the variable `unsorted` to `true`, as we've just made a change in the order of our dataset, and ensure that the item's new position doesn't put it in conflict with its new neighbors.

Now we've got a sorting algorithm, let's use it:

tablesort.js (excerpt)

```
var col = [], top, bottom;

if (th.className.match("asc")) { ❶
  top = bubbleSort(alpha, -1);
  bottom = bubbleSort(numeric, -1);
  th.className = th.className.replace(/asc/, "dsc");
} else { ❷
  top = bubbleSort(numeric, 1);
  bottom = bubbleSort(alpha, 1);
  if (th.className.match("dsc")) {
    th.className = th.className.replace(/dsc/, "asc");
  } else {
    th.className += "asc";
  }
}
```

- 1 First, we check to see the current state of our column. Is it sorted in ascending order? If it is, we'll call our bubble sort algorithm, requesting that it sort our array in descending order.
- 2 Otherwise, if the column's data is already sorted in descending order (or is unsorted), we request that it be sorted in ascending order.

We call `bubbleSort` twice because we've split our column data into two separate arrays, `alpha` and `numeric`. The results of the sort are then placed into two generic arrays, `top` and `bottom`. We use this approach because we can't be sure in advance whether we're going to be sorting in ascending or descending order. If the sort order is ascending, the `top` array will contain numeric data and the `bottom` array will contain alphanumeric data; when we're sorting in descending order, this assignment of array to data type is reversed. This approach should be fairly intuitive, given that once they're assigned, the contents of `top` will always appear at the top, and the contents of `bottom` will always appear at the bottom of our column.

Managing Heading States

Once the data's sorted, we set the `th` element's class name to either `asc` or `dsc` to reflect the column's current state. This will allow us to toggle the sort order back and forth if ever the user clicks on the same heading twice. Figure 1.4 shows the first column of our table in each of its toggle states.

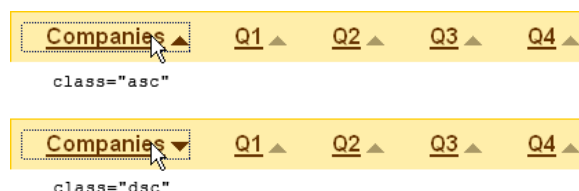


Figure 1.4. Heading states

Before we modify any headings, we need to make sure that the only column to have an `asc` or `dsc` class name is the one that was clicked. Each of the other columns needs to be reverted to its original, unsorted order—we do so by removing any `asc` or `dsc` class names that may previously have been assigned to those columns. In order to do this, we'll need a `TableSort`-level variable that will always remember the `th` element that belongs to the column that was last sorted. Let's go back and add a variable declaration called `this.lastSortedTh` to our `TableSort` function. We'll also write a small loop that will seek out any `asc` or `dsc` class names present in our HTML, and will store the last occurrence in `this.lastSortedTh`:

tablesort.js (excerpt)

```
function TableSort(id) {
  this.tbl = document.getElementById(id);
  this.lastSortedTh = null;
  if (this.tbl && this.tbl.nodeName == "TABLE") {
```

```

    var headings = this.tbl.tHead.rows[0].cells;
    for (var i=0; headings[i]; i++) {
        if (headings[i].className.match(/asc|dsc/)) {
            this.lastSortedTh = headings[i];
        }
    }
    this.makeSortable();
}
}

```

The variable `this.lastSortedTh` will now reflect any columns that are naturally sorted in the HTML; the `for` loop above sees to this by simply by reading the class names of the headings in the `thead`. In our example, even if the first click to sort a column occurred on a column other than the “Companies” column, our code would still be able to remove the “Companies” column’s `asc` value, because a reference to that column’s `th` is now held in `this.lastSortedTh`.

Here’s how we’ll clear the class names for previously sorted `th` elements:

tablesort.js (excerpt)

```

if (this.lastSortedTh && th != this.lastSortedTh) {
    this.lastSortedTh.className =
        this.lastSortedTh.className.replace(/dsc|asc/g, "");
}
this.lastSortedTh = th;

```

In the above code, we check to see whether a value is assigned to `this.lastSortedTh`. Then we verify that any value it *does* have is not simply a reference to the current column (we don’t want to clear the class names for the column we’re in the process of sorting!). Once we’re sure that this is indeed a valid column heading (and not the current one), we can go ahead and clear it using a simple regular expression.

Rearranging the Table

At this point in our script, we have two sorted arrays (named `top` and `bottom`), and a bunch of column headings that properly reflect the new sort state. All that’s left to do is to actually reorder the table’s contents:

tablesort.js (excerpt)

```

col = top.concat(bottom);
var tBody = this.tbl.tBodies[0];
for (var i=0; col[i]; i++) {
    tBody.appendChild(col[i].row);
}

```

The first thing we do is build a single array, `col1`, from the two that contain our sorted data. We do this by concatenating the contents of `bottom` to `top`, and placing the whole thing into the variable `col1`. Next, we loop over the contents of the `col1` array, taking each item's parent row and moving it to the bottom of the table's `tbody`. By doing this, we order the column's cells while keeping their relationships with the cells in other columns intact. Figure 1.5 demonstrates this process in action.

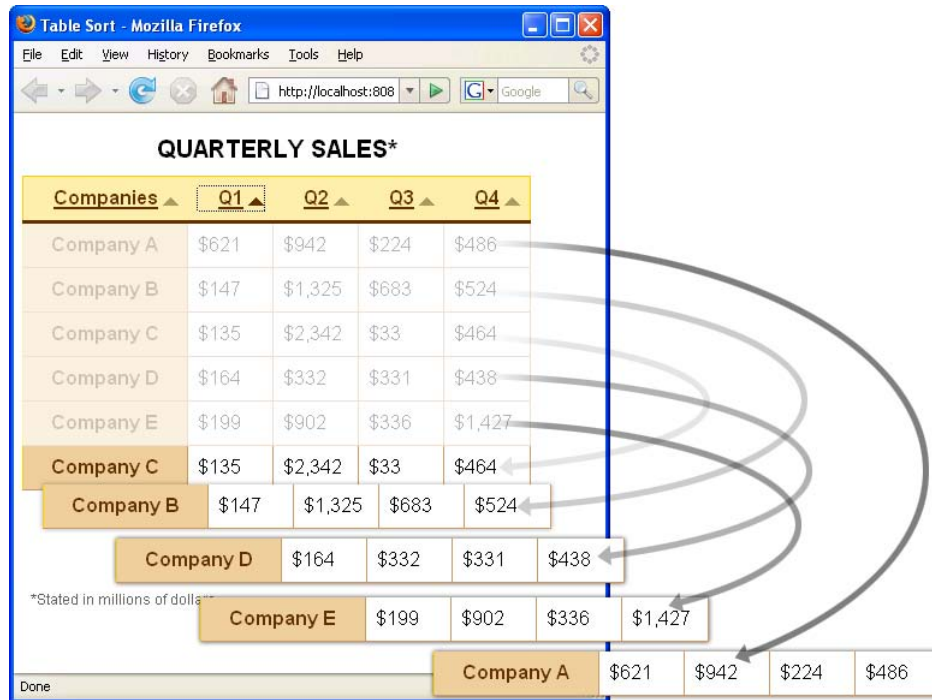


Figure 1.5. Rearranging the table

And with that final step, our script is complete! All that's left for us to do is to call `TableSort`. We do this by adding the following code to our document's head:

tablesort.html (excerpt)

```
<script type="text/javascript" src="tablesort.js"></script>
<script type="text/javascript">
window.onload = function () {
    var sales = new TableSort("sales");
} </script>
```

Though there are more optimal ways of doing it, for the sake of brevity I've used `window.onload` to call `TableSort` when the page loads. Remember that to make multiple tables sortable, all you need to do is create another instance of `TableSort` using a different table's ID. Here's our final script, in all its glory:

```

function TableSort(id) {
  this.tbl = document.getElementById(id);
  this.lastSortedTh = null;
  if (this.tbl && this.tbl.nodeName == "TABLE") {
    var headings = this.tbl.tHead.rows[0].cells;
    for (var i=0; headings[i]; i++) {
      if (headings[i].className.match(/asc|dsc/)) {
        this.lastSortedTh = headings[i];
      }
    }
    this.makeSortable();
  }
}

TableSort.prototype.makeSortable = function () {
  var headings = this.tbl.tHead.rows[0].cells;
  for (var i=0; headings[i]; i++) {
    headings[i].cIdx = i;
    var a = document.createElement("a");
    a.href = "#";
    a.innerHTML = headings[i].innerHTML;
    a.onclick = function (that) {
      return function () {
        that.sortCol(this);
        return false;
      }
    }(this);
    headings[i].innerHTML = "";
    headings[i].appendChild(a);
  }
}

TableSort.prototype.sortCol = function (el) {
  /*
   * Get cell data for column that is to be sorted from HTML table
   */
  var rows = this.tbl.rows;
  var alpha = [], numeric = [];
  var aIdx = 0, nIdx = 0;
  var th = el.parentNode;
  var cellIndex = th.cIdx;
  for (var i=1; rows[i]; i++) {
    var cell = rows[i].cells[cellIndex];
    var content =
      cell.textContent ? cell.textContent : cell.innerText;
    /*
     * Split data into two separate arrays, one for numeric content
     * and one for everything else (alphabetical). Store both the

```



```

    * actual data that will be used for comparison by the sort
    * algorithm (thus the need to parseFloat() the numeric data)
    * as well as a reference to the element's parent row. The row
    * reference will be used after the new order of content is
    * determined in order to actually reorder the HTML
    * table's rows.
    */
    var num = content.replace(/(\$|\,|\s)/g, "");
    if (parseFloat(num) == num) {
        numeric[nIdx++] = {
            value: Number(num),
            row: rows[i]
        }
    } else {
        alpha[aIdx++] = {
            value: content,
            row: rows[i]
        }
    }
}

/*
 * Sort according to direction (ascending or descending)
 */
var col = [], top, bottom;
if (th.className.match("asc")) {
    top = bubbleSort(alpha, -1);
    bottom = bubbleSort(numeric, -1);
    th.className = th.className.replace(/asc/, "dsc");
} else {
    top = bubbleSort(numeric, 1);
    bottom = bubbleSort(alpha, 1);
    if (th.className.match("dsc")) {
        th.className = th.className.replace(/dsc/, "asc");
    } else {
        th.className += "asc";
    }
}

/*
 * Clear asc/dsc class names from the last sorted column's th if
 * it isn't the same as the one that was just clicked
 */
if (this.lastSortedTh && th != this.lastSortedTh) {
    this.lastSortedTh.className =
        this.lastSortedTh.className.replace(/dsc|asc/g, "");
}
this.lastSortedTh = th;

/*

```

```

    * Reorder HTML table based on new order of data found in the
    * col array
    */
    col = top.concat(bottom);
    var tBody = this.tbl.tBodies[0];
    for (var i=0; col[i]; i++) {
        tBody.appendChild(col[i].row);
    }
}

function bubbleSort(arr, dir) {
    // Pre-calculate directional information
    var start, end;
    if (dir === 1) {
        start = 0;
        end = arr.length;
    } else if (dir === -1) {
        start = arr.length-1;
        end = -1;
    }

    // Bubble sort: http://en.wikipedia.org/wiki/Bubble_sort
    var unsorted = true;
    while (unsorted) {
        unsorted = false;
        for (var i=start; i!=end; i=i+dir) {
            if (arr[i+dir] && arr[i].value > arr[i+dir].value) {
                var a = arr[i];
                var b = arr[i+dir];
                var c = a;
                arr[i] = b;
                arr[i+dir] = c;
                unsorted = true;
            }
        }
    }
    return arr;
}

```

Creating Draggable Columns

A feature that's often desired by those working with tabular data in desktop applications is the ability to move a table's columns around in order to get a better look at its data. For example, you may want to compare the values in the first and last columns of a table. Being able to move those columns next to each other makes the task much easier.

But why should desktop applications have all the fun? We can create draggable columns on the Web too—Figure 1.6 shows an example of a simple HTML table that we’ll add this functionality to.

As with our sorting script, let’s make this functionality as accessible as possible by:

- ensuring it works with and without a mouse
- ensuring it works with multiple tables on the one page
- making it super-easy to implement

A Simple Table - Mozilla Firefox

File Edit View History Bookmarks Tools Help

file:///C:/Document: Google

QUARTERLY SALES*

Companies	Q1	Q2	Q3	Q4
Company A	\$621	\$942	\$224	\$486
Company B	\$147	\$1,325	\$683	\$524
Company C	\$135	\$342	\$33	\$464
Company D	\$164	\$332	\$331	\$438
Company E	\$199	\$902	\$336	\$1,427

*Stated in millions of dollars

Done

Want to move a column?

Figure 1.6. Want to move a column?

Let’s address the third point in that list. All that will be required to implement the draggable column functionality is the addition of a single line of code to the page, as in our sorting example in the previous section:

```

window.onload = function () {
    var sales = new ColumnDrag("sales");
}

```

Making the Table's Columns Draggable

Taking the lead from our table sort example, we first need to create a JavaScript function that we can instantiate. We’ll call this function `ColumnDrag`:

columnDrag.js (excerpt)

```
function ColumnDrag(id) {
  this.tbl = document.getElementById(id);
  if (this.tbl && this.tbl.nodeName == "TABLE") {
    this.state = null;
    this.prevX = null;
    this.cols = this.tbl.getElementsByTagName("col");
    this.makeDraggable();
  }
}
```

In the code above, the first thing that we do is try to access the table whose columns are to be made draggable. If a table element with the id we've specified exists, we know that we're in business.

Next, we set a few variables that we'll be using later on. Note that unlike the `tBodies`, `rows`, and `cells` elements, there isn't a DOM reference for the `col` element. So instead of spending precious CPU cycles calling `getElementsByTagName` every time we want to access one of the newly added `col` elements below, we've used it here once, and stored the references that it returned for later use. Here's where the `col` elements fit into our markup:

columnDrag.html (excerpt)

```
<table id="sales" summary="Quarterly sales figures for competing
companies. The figures are stated in millions of dollars.">
  <caption>Quarterly Sales*</caption>
  <col />
  <col />
  <col />
  <col />
  <col />
  <thead>
  :
```

The `col` element is used as a convenient way to apply styles to columns of data in a table—you can add a class to a `col` instead of adding classes to each individual `td` in a column. We'll use the `col` element to highlight the column that's being dragged.

The Phantom Column

The approach we'll use to move our columns around is very similar to the one we used to reorganize our table in the table sort example. However, in this case we're not moving rows, but cells. We'll also be inserting them into their new positions, rather than always appending our cells to the end of the collection. To achieve this aim, we'll use the `insertBefore` method, which takes two parameters: the node that's to be inserted, and the node before which it will be inserted.

Now, normally this method is used to insert a newly created node, like so:

```
var para = document.getElementById("foo");
var newPara = document.createElement("p");
para.parentNode.insertBefore(newPara, para);
```

However, the `insertBefore` method isn't limited to just inserting new elements into the DOM—we can also use it to shuffle existing elements around within the DOM. For this example, we'll be doing something like this:

```
row.insertBefore(row.cells[a], row.cells[b]);
```

Suppose, for example, that we wanted to move the third cell in our row to the end of our collection of cells. How should we specify where the node is to be inserted? Some browsers, such as Firefox, allow us to refer to “the last cell plus one.” Others (in particular, Internet Explorer) won't allow this, instead telling us that “the last cell plus one” doesn't exist.

To work around this problem, we'll insert a phantom column of cells at the end of the table. These cells will be hidden, and will serve only as a valid reference before which we can always insert cells:

columndrag.js (excerpt)

```
ColumnDrag.prototype.makeDraggable = function () {
  for (var i=0; this.tbl.rows[i]; i++) {
    var td = document.createElement("td");
    td.style.display = "none";
    this.tbl.rows[i].appendChild(td);
  }
  :
}
```

Accessible Dragging

As before, we want to make this functionality as accessible as possible—and that means making it work without a mouse. So, once again, we'll introduce anchors in order to allow the user to tab from one `th` to the other:

columndrag.js (excerpt)

```
var headings = this.tbl.tHead.rows[0].cells;
for (var i=0; headings[i]; i++) {
  headings[i].cIdx = i;
  var a = document.createElement("a");
  a.href = "#";
  a.innerHTML = "&larr; "+headings[i].innerHTML+ " &rarr;";
```

```

a.onclick = function () {
    return false;
}
headings[i].className += " draggable";
:

```

This time, however, the anchor's `onclick` event handler does nothing but return `false`. This ensures that clicking an anchor won't cause a new entry to be added to our browser's history—an action that would scroll the browser window to the top of the page. In this particular example, we don't need to worry about this issue—we're only using a small table at the top of the page. However, if your table was located in the middle or at the bottom of a very long page, this behavior could be disastrous.

You'll notice that I've added a property named `cIdx` to our `<th>` tags. As with our table sort example, this property just numbers the cells—a task that we'd normally leave to the built-in property `cellIndex`. However, because Safari 2.0.4 always returns 0 for `cellIndex`, we'll emulate this behavior with our own `cIdx` property.

I've taken the liberty of adding left and right arrow characters (the `←` and `→` entities, respectively) to either side of the `th` element's existing content. These characters act as a visual cue to the user that the arrow keys can be used to drag columns within the table.

I've also added a small graphic to the upper right-hand corner of each `th` cell—a “grip” that indicates to mouse users that the column is draggable. I've implemented this graphic as a background image via the `draggable` class name in the style sheet. Some may consider using both of these cues together to be overkill, but I think they work quite well—we'll let the usability experts argue over that one! Figure 1.7 shows our table with both of these visual flourishes in place.



Alternatives to the Arrow Keys

Under certain circumstances, you may find that the arrow keys behave unreliably in certain browsers. If you run into this problem, consider using keys other than the arrow keys—for example, **n** and **p** could be used for **N**ext and **P**revious. If you follow such an approach, be sure to point out these keyboard shortcuts to your users, or your hard work may go unused!

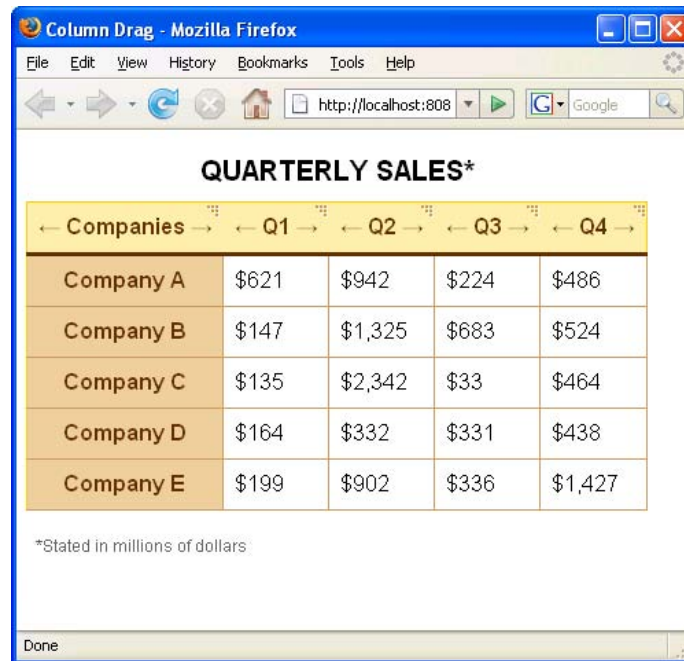


Figure 1.7. Providing visual cues for dragging

Event Handling

Our next step is to wire up the mousedown, mousemove, mouseup, and mouseout events. This is a relatively straightforward process—once again, we’re using the scope correction technique that we employed in the table sort example, whereby we point the `this` keyword to our instance of `ColumnDrag`, rather than the element that triggered the event. We also pass the event object `e` to the functions that will be handling our events:

columndrag.js (excerpt)

```

headings[i].mousedown = function (that) {
  return function (e) {
    that.mousedown(e);
    return false;
  }
}(this);
document.onmousemove = function (that) {
  return function (e) {
    that.mousemove(e);
    return false;
  }
}(this);
document.onmouseup = function (that) {
  return function () {
    var e = that.clearAllHeadings();
    if (e) that.mouseup(e);
  }
}

```

```

    }
  }(this);
  document.onmouseout = function (that) {
    return function (e) {
      e = e ? e : window.event;
      related = e.relatedTarget ? e.relatedTarget : e.toElement;
      if (related == null) {
        var e = that.clearAllHeadings();
        if (e) that.mouseup(e);
      }
    }
  }(this);

  a.onkeyup = function (that) {
    return function (e) {
      that.keyup(e);
      return false;
    }
  }(this);

```

The `onmousemove`, `onmouseup`, and `onmouseout` event handlers are set up to trap the event no matter where it occurs in the document. We've taken this approach to ensure that the drag functionality isn't limited to the area that the `th` elements occupy. One drawback to this approach, however, is that neither the `onmouseup` or the `onmouseout` event handlers know which `th` is currently being dragged, so these event handlers can't reset the event if the user should drag the mouse out of the browser window or release it somewhere other than on a `<th>`. In order to fix that, we'll use a function that we've called `clearAllHeadings`. This function will cycle through all the headings, clear them of the `down` class name, and return a reference to the `th` so that we can pass it to the `mouseup` function. We'll flesh out the `clearAllHeadings` function in just a minute.

Earlier, we set our anchor's `onclick` event to do nothing but return `false`. Here, we're giving the anchor an `onkeyup` event handler—this will allow us to trap the left and right arrow key events for accessibility purposes.

We also add a `hover` class to, or remove it from, our `th` elements whenever a `mouseover` or a `mouseout` event is triggered on them, respectively. We do this because anchors are the only elements for which Internet Explorer 6 supports the CSS `:hover` pseudo-class:

columnDrag.js (excerpt)

```

    headings[i].onmouseover = addHover;
    headings[i].onmouseout = removeHover;
    headings[i].innerHTML = "";
    headings[i].appendChild(a);
  }
}

```


Here's that `clearAllHeadings` function that I promised you earlier. After a heading has begun to be dragged, this function will be called if the mouse leaves the browser window, or if the mouse button is released:

```
ColumnDrag.prototype.clearAllHeadings = function () {
  var e = false;
  for (var i=0; this.cols[i]; i++) {
    var th = this.tbl.tHead.rows[0].cells[i];
    if (th.className.match(/down/)) {
      e = {target: th};
    }
  }
  return e;
}
```

Here are the `addHover` and `removeHover` functions:

columndrag.js (excerpt)

```
addHover = function () {
  this.className += " hover";
}
removeHover = function () {
  this.className = this.className.replace(/ hover/g, "");
}
```

The `addHover` function simply adds a `hover` class name, preceded by a space (to prevent a conflict with any existing class names). The `removeHover` function removes all occurrences of the `hover` class name from an element.

In both functions, we leave the `this` keyword alone and avoid trying to correct the scope. In this example, `this` behaves exactly the way we need it to: it refers to the element whose class name we want to modify. Also note that we don't need a reference to our instance of `ColumnDrag`, which is why we haven't bothered adding these functions to `ColumnDrag` via the prototype object.

The onmousedown Event Handler

Our `onmousedown` event handler reads as follows:

columndrag.js (excerpt)

```
ColumnDrag.prototype.mousedown = function (e) {
  e = e ? e : window.event;
  var elm = e.target? e.target : e.srcElement;
  elm = elm.nodeName == "A" ? elm.parentNode : elm;

  this.state = "drag";
}
```

```

elm.className += " down";
this.cols[elm.cIdx].className = "drag";
this.from = elm;
operaRefresh();
}

```

The event object that I mentioned earlier is an object created by the browser when an event is triggered. It contains information that's pertinent to the event, such as the element that triggered it, the mouse's *x* and *y* coordinates at the moment it was triggered, and so on. Unfortunately, while some browsers pass the event object as a parameter to the event handler, others provide access to this object differently. For example, in the code above, we anticipate receiving the event object as the argument *e* that's passed to the event handler. However, we also prepare a fallback approach whereby if the code finds that *e* has not been set, we set it via the global `window.event` object, which is how Internet Explorer delivers this event information. Once again, the ternary operator comes in handy—we use it here to set the object into our *e* variable regardless of where it comes from.

The cross-browser inconsistencies don't end there, though. Passed along inside the event object is a reference to the object that triggered it. Some browsers use the `target` property name to store this reference, while others (I'm looking at you, Internet Explorer!) call it `srcElement`. We normalize this behavior as well, using another ternary operator, and store the result in the *elm* variable.

If the user clicks on one of our wrapper anchors, the event for this mouse click will bubble up and trigger the parent *th* element's `onmousedown` event. When that occurs, *elm* will refer to an anchor, rather than the *th* we require. We need to correct this; otherwise, we risk having an incorrect reference passed to the `mousedown` function. So, if *elm*'s `nodeName` value is `A` (meaning that it's an anchor element), we know that we need its parent. So we introduce yet another ternary operator to choose between *elm* and its parent node.

You'll also notice a call to the `operaRefresh` function in our code. This is an unfortunate but necessary evil, because the latest version of Opera (9.23 at the time of writing) doesn't refresh the affected cells of a table when a class name is added to or changed on a `col` element. All this function does is change the body element's position from the browser's default of `static` to `relative` and back again, forcing a refresh. Here's what the `operaRefresh` function looks like:

columnDrag.js (excerpt)

```

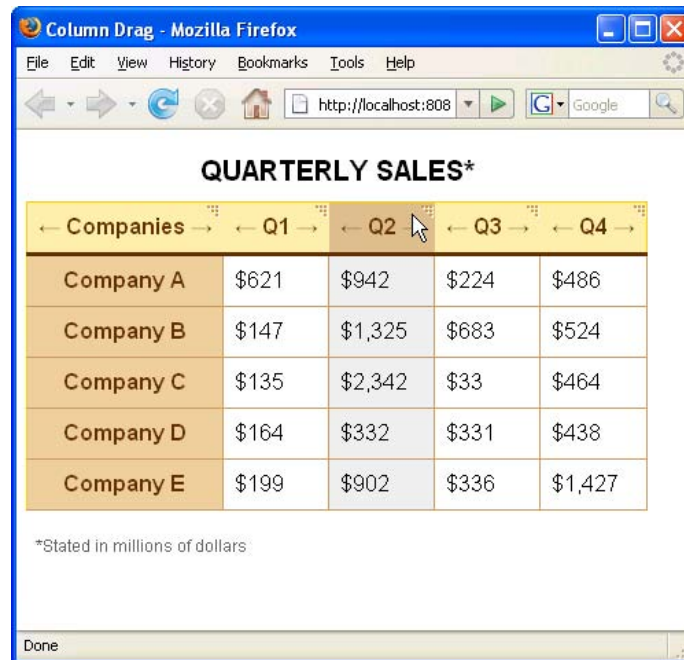
operaRefresh = function () {
    document.body.style.position = "relative";
    document.body.style.position = "static";
}

```

Depending on the final value that you require for your style sheet, you may need to use different values here—it might seem odd, but the important thing is that the body’s position changes between two different values in order for Opera to play ball.

All this function does is change the body element’s position from the browser’s default of `static` to `relative`, and back again, which forces a refresh. Depending on the position values that you’ve used in your style sheet, you may want the final value to be something else.

Figure 1.8 shows what our column looks like once the user has clicked on the “Q2” heading cell.



Companies	Q1	Q2	Q3	Q4
Company A	\$621	\$942	\$224	\$486
Company B	\$147	\$1,325	\$683	\$524
Company C	\$135	\$2,342	\$33	\$464
Company D	\$164	\$332	\$331	\$438
Company E	\$199	\$902	\$336	\$1,427

*Stated in millions of dollars

Figure 1.8. mousedown in action

The onmousemove Event Handler

As I mentioned earlier, the basic principle behind the column drag functionality is to take the currently selected column of cells and reinsert them elsewhere in the table. As we’re using the `insertBefore` function, we don’t need to worry about shifting the surrounding cells into their new positions.

The `mousemove` function is called every time the cursor moves over a `th` cell in the table’s `thead`. Here’s the first part of the code for this function:

columnDrag.js (excerpt)

```
ColumnDrag.prototype.mousemove = function (e) {
  e = e ? e : window.event;
  var x = e.clientX ? e.clientX : e.pageX;
```

```
var elm = e.target? e.target : e.srcElement;
if (this.state == "drag" && elm != this.from) {
  :
}
```

First, we normalize the event object, the mouse's current x coordinate, and the element that triggered the `onmousemove` event. Then we make sure that we're currently in `drag` mode—if the value of `this.state` isn't set to `drag`, then the user hasn't kept the mouse button pressed down and is simply hovering over the `th` elements. If that's the case, we don't need to do anything. But if the value of `this.state` is set to `drag`, we proceed with dragging the column:

columnDrag.js (excerpt)

```
var from = this.from.cIdx; ❶
var to = elm.cIdx; ❷

if ((from > to && x < this.prevX)
    || (from < to && x > this.prevX)) { ❸
```

- ❶ In order to know where we're dragging *from*, we first grab the cell index value (`cIdx`) of the `th` element that we stored in the `this.from` variable back in our `onmousedown` function.
- ❷ In order to know where we're dragging *to*, we grab the `cIdx` value of the `th` we're currently hovering over.
- ❸ Now, because the columns in our table can have wildly different widths, we need to make sure that the direction in which we're moving the column and the direction in which the mouse is moving are synchronized. If we don't perform this check, the moment a wider `th` takes the place of the one we're dragging, it will *become* the cell we're dragging, as it will end up under the cursor, and will therefore replace the object stored in `elm`. Our code will then try and swap the original cell back, causing an unsightly flickering effect.

We therefore check the relationship between the `from` and `to` values and the cursor's current and previous x coordinates. If `from` is greater than `to`, and the mouse's current x coordinate is less than it was the previous time we called `mousemove`, we proceed with the drag. Likewise, if `from` is less than `to`, and the x coordinate is greater than the previous value, then again we have the green light to proceed.

Before we actually move the cells, we must alter the class names of the `col` elements representing the positions that we're dragging cells from and to. To achieve this, we use the `from` and `to` variables that we created a few lines earlier. Since we're not using the `col` elements for anything other than to signify whether a column is being dragged or not, we can safely clear the existing `drag` value from the `from` column by assigning it a value of `" "`. We can then assign the `drag` class name to the `to` column:

columndrag.js (excerpt)

```
this.cols[from].className = "";
this.cols[to].className = "drag";
```

Highlighting the column in the style sheet as it moves, as pictured in Figure 1.8, helps users follow what's happening while they're dragging the column.

Since we're using `insertBefore` to move our cells around, we might need to make a slight adjustment to our `to` value. If we're moving forward (that is, from left to right) we need to increment the value of `to` by one, like so:

columndrag.js (excerpt)

```
if (from < to) to++;
```

This alteration is necessary because we need to insert our `from` cell before the cell that comes *after* the `to` cell, as Figure 1.9 reveals.

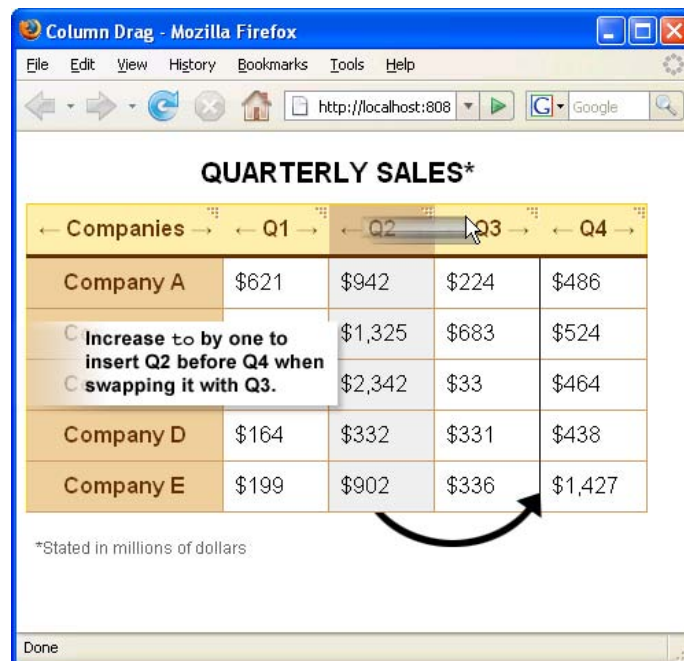


Figure 1.9. Increasing `to` by one

Now that we've worked out our references, all that's left to do is cycle through all of our cells and move them:

columndrag.js (excerpt)

```
var rows = this.tbl.rows;
for (var i=0; rows[i]; i++) {
    rows[i].insertBefore(rows[i].cells[from], rows[i].cells[to]);
```

Next, we update our cIdx values, as they'll be out of sync now that we've reordered our cells:

```
var headings = this.tbl.tHead.rows[0].cells;
for (var i=0; headings[i]; i++) {
    headings[i].cIdx = i;
}
}
}
}
this.prevX = x;
```

The last thing that we do before exiting mousemove is to register the current x coordinate as the previous one in this.prevX. That way, the next time we enter mousemove, we'll know in which direction the mouse is moving.

The onmouseup Event Handler

Once the users are happy with the position of the dragged column, they'll let go of the mouse button. That's when the onmouseup event will be triggered. Here's the mouseup function that's called by this event:

columndrag.js (excerpt)

```
ColumnDrag.prototype.mouseup = function (e) {
    e = e ? e : window.event;
    var elm = e.target? e.target : e.srcElement;
    elm = elm.nodeName == "A" ? elm.parentNode : elm;
    this.state = null; ❶
    var col = this.cols[elm.cellIndex];
    col.className = "dropped"; ❷
    operaRefresh();
    window.setTimeout(function (cols, el) {
        return function () {
            el.className = el.className.replace(/ down/g, "");
            for (var i=0; cols[i]; i++) {
                cols[i].className = "";
            }
            operaRefresh();
        }
    })(this.cols, this.from), 1000); ❸
}
```

`onmouseup`'s purpose is to reset all of the variables that have to do with the drag.

- ❶ We set `this.state` to null right away. This tells any future call to `mousemove` functions not to drag anything (that is, of course, until the mouse button is pressed again).
- ❷ We then write the value dropped to the `col` element that represents the final resting place of our dragged column. This allows us to change the column's color, which indicates visually to the user that its state has changed.
- ❸ We set up a one-second delay, using `window.setTimeout` to reset both the `col`'s and the `th`'s class names. This delay allows us to linger a little on the column that was dropped, making for a better user experience.

You'll notice our `operaRefresh` function pops up again in two places throughout this code. It ensures that the newly modified class names on the table's `col` elements are applied properly in the Opera browser.

Dragging Columns without a Mouse

Earlier, I wrapped the contents of our `th` elements with anchors for accessibility purposes. I also gave them an `onkeyup` event handler. Normally, we'd write code specifically to handle key releases, but in the spirit of reuse, let's write code that will convert left and right arrow button presses into left and right mouse drags. That way, we can reuse the column drag code that we just finished writing for the mouse. In short, we'll be using the keyboard to impersonate the mouse and the events that a mouse would trigger.

The only parameter that's ever passed to `mousedown`, `mousemove`, and `mouseup` is the event object. We can retrieve everything we need from it and a few variables that are set in our instance of `ColumnDrag`.



onkeyup versus onkeydown

The reason I capture the `onkeyup` event, and not `onkeydown` or `onkeypress`, is because `onkeyup` only fires once, whereas the other events continue firing so long as the key is kept down. This can make it really difficult to accurately move a column to the right position if the user presses the key for too long. Using `onkeyup` ensures that one action equals one move.

All we have to do now, in order to pretend we're the mouse when we're calling those functions, is to set those variables manually, and pass our own fake event object, like so:

columnndrag.js (excerpt)

```
ColumnDrag.prototype.keyup = function (e) {
  e = e ? e : window.event;
  var elm = e.target ? e.target : e.srcElement;
  var a = elm;
  elm = elm.parentNode;
  var headings = this.tbl.tHead.rows[0].cells;
```

As always, our first step is to normalize our event object and the clicked element into `e` and `elm`, respectively. We then make a backup of `elm` in `a` and change `elm` to point to its parent node. This will change our `elm` variable from an anchor into a `th` element.

Next, we capture the events for the left and right key releases:

columnndrag.js (excerpt)

```
switch (e.keyCode){ ❶
  case 37:
    this.mousedown({target:elm}); ❷
    var prevCellIdx = elm.cIdx==0 ? 0 : elm.cellIndex-1;
    this.prevX = 2;
    this.mousemove(
      {
        target: headings[prevCellIdx],
        clientX: 1
      }
    );
    this.mouseup({target: elm});
    a.focus();
    break;
  case 39:
    this.mousedown({target:elm});
    var nextCellIdx = elm.cellIndex==headings.length-2 ?
      headings.length-2 : elm.cellIndex+1;
    this.prevX = 0; ❸
    this.mousemove(
      {
        target: headings[nextCellIdx],
        clientX: 1
      }
    );
    this.mouseup({target: elm});
    a.focus();
```



```

        break;
    }
}

```

Let's look at what's going on in the above code:

- ❶ We create a `switch` block to compare the value of the event object's `keyCode` property with the key codes 37 and 39, which represent the left and right arrow keys respectively. In both cases, we call `mousedown` and pass it `elm` as the value for `target`.
- ❷ Note that because we're passing our own fake event object, we need to assign a value for `target` only—not for `srcElement`—regardless of which browser's being used. The same advice applies when we set `clientX` in the object that we pass `mousemove`.
- ❸ We need to set a few extra values when calling `mousemove`. Specifically, we need to set `this.prevX` and `clientX` so that one value is greater than the other, depending on which way we want to pretend the mouse is moving. If `clientX` is greater than `this.prevX`, for example, the mouse is moving to the right.

When we pass our event handler a value for `target`, we need to make sure we aren't out of bounds. This task is fairly simple when the column's moving left: all we need to do is make sure that our `cellIndex` value doesn't fall below zero. Moving the column right, however, means checking against the number of cells in the row minus two—one for the fact that `length` returns a one-based result (while the collection of cells is zero-based just like an array) and one extra, to account for the phantom cells we inserted earlier.



Object Literal Notation

We've just seen an example in which an object was created using **object literal** notation. A simple example of this is: `var myObj = {};`

Here's a slightly more complex example:

```

var groceryList = {
  bananas: 0.49,
  apples: 1.39,
  milk: 5.99
};

```

In the above example, the three variables that are declared are effectively public properties of the `groceryList` object. More details about the syntax that's available to those using an object oriented approach in JavaScript can be found in Chapter 5.

Lastly, we call `mouseup` and pass it our `th`, which is now in its new position. We finish off by setting `focus` on its anchor, which sets us up for the next key release.

Now, we add our instantiation of `ColumnDrag` to our document's head, and we're done:

columndrag.html (excerpt)

```
<script type="text/javascript" src="columndrag.js"></script>
<script type="text/javascript">
window.onload = function () {
    var sales = new ColumnDrag("sales");
}
</script>
```

Here's our final code in full:

columndrag.js

```
function ColumnDrag(id) {
    this.tbl = document.getElementById(id);
    if (this.tbl && this.tbl.nodeName == "TABLE") {
        this.state = null;
        this.prevX = null;
        this.cols = this.tbl.getElementsByTagName("col");
        this.makeDraggable();
    }
}

ColumnDrag.prototype.makeDraggable = function () {
    // Add trailing text node for IE
    for (var i=0; this.tbl.rows[i]; i++) {
        var td = document.createElement("td");
        td.style.display = "none";
        this.tbl.rows[i].appendChild(td);
    }

    // Wire up headings
    var headings = this.tbl.tHead.rows[0].cells;
    for (var i=0; headings[i]; i++) {
        headings[i].cIdx = i; // Safari 2.0.4 "cellIndex always equals 0" workaround

        var a = document.createElement("a");
        a.href = "#";
        a.innerHTML = "&larr; " + headings[i].innerHTML + " &rarr;";
        a.onclick = function () {
            return false;
        }

        headings[i].className += " draggable";
    }
}
```

```

        headings[i].onmousedown = function (that) {
            return function (e) {
                that.mousedown(e);
                return false;
            }
        }(this);
        document.onmousemove = function (that) {
            return function (e) {
                that.mousemove(e);
                return false;
            }
        }(this);
        document.onmouseup = function (that) {
            return function () {
                var e = that.clearAllHeadings();
                if (e) that.mouseup(e);
            }
        }(this);
        document.onmouseout = function (that) {
            return function (e) {
                e = e ? e : window.event;
                related = e.relatedTarget ? e.relatedTarget : e.toElement;
                if (related == null) {
                    var e = that.clearAllHeadings();
                    if (e) that.mouseup(e);
                }
            }
        }(this);
        a.onkeyup = function (that) {
            return function (e) {
                that.keyup(e);
                return false;
            }
        }(this);
        headings[i].onmouseover = addHover;
        headings[i].onmouseout = removeHover;

        headings[i].innerHTML = "";
        headings[i].appendChild(a);
    }
}

ColumnDrag.prototype.clearAllHeadings = function () {
    var e = false;
    for (var i=0; this.cols[i]; i++) {
        var th = this.tbl.tHead.rows[0].cells[i];
        if (th.className.match(/down/)) {
            e = {target: th};
        }
    }
}

```

```

    }
    return e;
}

ColumnDrag.prototype.mousedown = function (e) {
    e = e ? e : window.event;
    var elm = e.target? e.target : e.srcElement;
    elm = elm.nodeName == "A" ? elm.parentNode : elm;

    // set state and clicked "from" element
    this.state = "drag";
    elm.className += " down";
    this.cols[elm.cIdx].className = "drag";
    this.from = elm;
    operaRefresh();
}

ColumnDrag.prototype.mousemove = function (e) {
    e = e ? e : window.event;
    var x = e.clientX ? e.clientX : e.pageX;
    var elm = e.target? e.target : e.srcElement;

    if (this.state == "drag" && elm != this.from) {
        var from = this.from.cIdx;
        var to = elm.cIdx;

        // make sure that mouse is moving in same dir as swap (to avoid
        // swap flickering)
        if ((from > to && x < this.prevX) || (from < to && x > this.prevX)) {

            // highlight column
            this.cols[from].className = "";
            this.cols[to].className = "drag";

            // increase 'to' by one if direction is positive because we're inserting
            // 'before' and so we have to refer to the target columns neighbor
            if (from < to) to++;

            // shift all cells belonging to head
            var rows = this.tbl.rows;
            for (var i=0; rows[i]; i++) {
                rows[i].insertBefore(rows[i].cells[from], rows[i].cells[to]);
            }

            // update cIdx value (fix for Safari 2.0.4 "cellIndex always equals 0" bug)
            var headings = this.tbl.tHead.rows[0].cells;
            for (var i=0; headings[i]; i++) {
                headings[i].cIdx = i;
            }
        }
    }
}

```

```

    }
    this.prevX = x;
}

ColumnDrag.prototype.mouseup = function (e) {
    e = e ? e : window.event;
    var elm = e.target ? e.target : e.srcElement;
    elm = elm.nodeName == "A" ? elm.parentNode : elm;

    this.state = null;
    var col = this.cols[elm.cIdx];
    col.className = "dropped";
    operaRefresh();
    window.setTimeout(function (that) {
        return function () {
            that.from.className = that.from.className.replace(/ down/g, "");
            for (var i=0; that.cols[i]; i++) {
                that.cols[i].className = ""; // loop over all cols to avoid odd sized
            } // column conflicts
            operaRefresh();
        }
    })(this), 1000);
}

ColumnDrag.prototype.keyup = function (e) {
    e = e ? e : window.event;
    var elm = e.target ? e.target : e.srcElement;
    var a = elm;
    elm = elm.parentNode;
    var headings = this.tbl.tHead.rows[0].cells;

    switch (e.keyCode){
        case 37: // left
            this.mousedown({target:elm});

            var prevCellIdx = elm.cIdx == 0 ? 0 : elm.cIdx - 1;
            this.prevX = 2;
            this.mousemove(
                {
                    target: headings[prevCellIdx],
                    clientX: 1
                }
            );

            this.mouseup({target: elm});

            a.focus();
            break;

        case 39: // right

```

```

        this.mousedown({target:elm});

        // -2 for IE fix phantom TDs
        var nextCellIdx =
            elm.cIdx == headings.length-2 ? headings.length-2 : elm.cIdx + 1;
        this.prevX = 0;

        this.mousemove(
            {
                target: headings[nextCellIdx],
                clientX: 1
            }
        );
        this.mouseup({target: elm});

        a.focus();
        break;
    }
}

addHover = function () {
    this.className += " hover";
}

removeHover = function () {
    this.className = this.className.replace(/ hover/, "");
}

operaRefresh = function () {
    document.body.style.position = "relative";
    document.body.style.position = "static";
}

```

Summary

I hope you've enjoyed reading this chapter as much as I've enjoyed writing it. In putting together these examples I've tried not only to show you how to manipulate tables through JavaScript, but also how to write efficient, optimized, and reusable JavaScript code.

In this chapter, we've learned how to access a table, its rows, cells, and various groups thereof. We've covered techniques for managing columns of data, even though sorting cells and dragging columns is not a native functionality of either HTML or JavaScript. And we've taken functionality that's traditionally considered to be "mouse only" and made it work with the keyboard as well.

I hope that at least some of the concepts I've shared here, such as regular expressions, sorting algorithms, event handlers, objects, and of course tables, will inspire you to dig deeper and build your own wild and crazy JavaScript apps!

Chapter 6

Building a 3D Maze with CSS and JavaScript

In this chapter we'll look at a technique for using CSS and JavaScript to build a first-person-perspective maze, in homage to old-school adventure games like *Dungeon Master*¹ and *Doom*.²

In truth, the scripting involved is fairly complex, and it won't be possible for me to spell out every nuance of the code in this single chapter. In fact, I won't even list every method used in the script, as some of them are quite long. What I can do, though, is introduce you to the principles of creating shapes and perspective with CSS, and the task of using JavaScript to generate those shapes on demand to create a dynamic, three-dimensional perspective from a static, two-dimensional map.

The script, and all of its components, are included in the book's code archive. All the code is robustly commented, so you should find it easy to follow. I recommend that you have it available to view as you read, so that you can refer to it as we go along.

Before we dive into a discussion of how it's built, let's take a look at the final result—it's shown in Figure 6.1.

¹ [http://en.wikipedia.org/wiki/Dungeon_Master_\(computer_game\)](http://en.wikipedia.org/wiki/Dungeon_Master_(computer_game))

² <http://en.wikipedia.org/wiki/Doom>

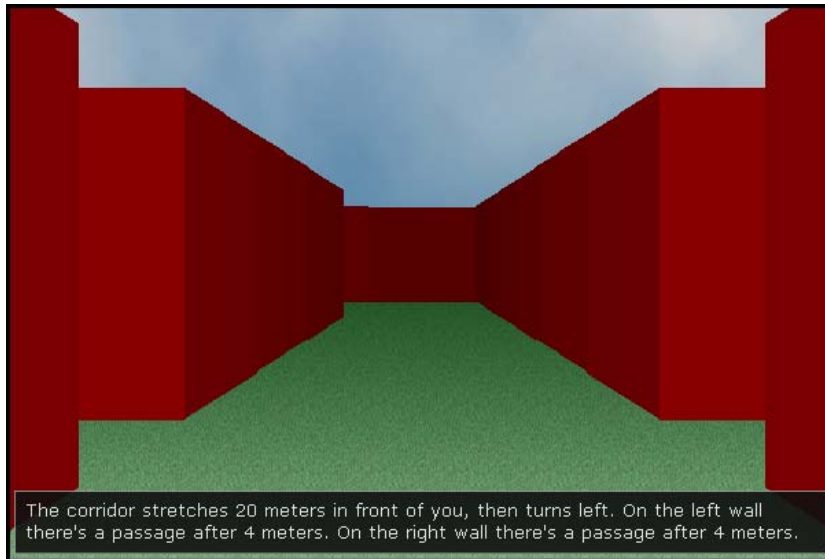


Figure 6.1. A view inside the finished maze

That screenshot was taken with Opera, in which this script was originally developed, and it also works as intended in Firefox, Safari, and Internet Explorer 7. IE 6, however, is not fully supported: the game works, but it looks poor because IE 6 doesn't have all the CSS support we need (most notably, it lacks support for transparent borders).

I should also point out, in case it crosses your mind, that what we're doing here has no practical use. In fact, it could be argued that we're not really using the right technology for the job. I made this maze because I wanted to see if it was possible—to push the envelope a little in terms of what can be done with JavaScript and CSS. But we're right at the edge of what's reasonable, and maybe Flash or SVG would be better suited to building a game like this.

But hey—why climb a mountain? Because it's there!

Basic Principles

In 2001, Tantek Çelik published a technique for creating shapes using the interactions between CSS borders.³ We're going to use that technique to make a bunch of right-angle triangles.

Why triangles, I hear you ask? Well, because once you can render a triangle, you can render any polygon that you like. By combining triangles with the rectangles that we've always been able to render (using a good old `div` and the `background-color` property), we can create the walls of our maze and contribute to the sense of perspective. As you'll see, we'll draw these walls by slicing the player's view up into a number of columns.

³ <http://tantek.com/CSS/Examples/polygons.html>

We'll also need a floor plan for our maze, and a handful of methods for dynamically converting that floor plan into the polygons that represent the walls of our maze.

Making Triangles

If an element has a very thick border (say 50px), and adjacent borders have different colors, the intersection of those borders creates a diagonal line, as Figure 6.2 illustrates.

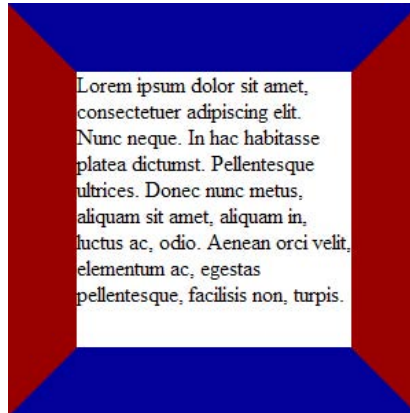


Figure 6.2. Making diagonal lines from CSS borders

That example is simply a `div` element to which the following CSS rules are applied:

```
width: 200px;
height: 200px;
border: 50px solid #900;
border-color: #009 #900;
```

To render a triangle, we don't actually need the contents of that `div`—we only need its borders. So let's remove the text, and reduce the width and height values to zero. What we're left with is the image shown in Figure 6.3.

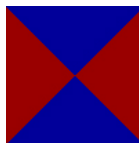


Figure 6.3. Making triangles from CSS borders

Here's the CSS that achieves that effect:

```
width: 0;
border: 50px solid #900;
border-color: #009 #900;
```

If we were to vary the relative border widths (applying, say, 50px on the left border and 25px on the top), we could create triangles with various angles. By setting the color of one of the borders to transparent, the diagonal line from the solid border stands alone, as Figure 6.4 reveals.



Figure 6.4. Creating diagonal lines using transparent adjacent borders

Now, if we wrap a second `div` element around the first, we'll be able to extract a single, discreet triangle. We can achieve this by:

1. applying `position: relative` to the outer container
2. applying `position: absolute` to the inner element
3. clipping the inner element

Clipped elements are required to have absolute positioning,⁴ so the relative positioning on the container provides a positioning context for the inner element, as Figure 6.5 shows.



Figure 6.5. Extracting a single triangle using CSS `clip`

The code that produces Figure 6.5 is still very simple. Here's the HTML:

```
<div id="triangle">
  <div></div>
</div>
```

And here's the CSS:

```
#triangle
{
  border: 2px solid #999;
  position: relative;
  width: 50px;
  height: 25px;
}
#triangle > div
{
  border-style: solid;
  border-color: transparent #900;
  border-width: 25px 50px;
  position: absolute;
  left: 0;
```

⁴ <http://www.w3.org/TR/CSS21/visufx.html#propdef-clip>

```
top: 0;
clip: rect(0, 50px, 25px 0);
}
```

Clipping and positioning is the crux of our ability to create discreet shapes using CSS. If we removed the `clip`, we'd get the result shown in Figure 6.6.



Figure 6.6. The unclipped element hangs outside its parent

You can see that by varying the `clip` and `position` properties on the inner element, we control which part of it is shown, and hence which of the triangles will be visible. If we wanted the bottom-right triangle, we would apply these values:

```
left: -50px;
top: -25px;
clip: rect(25px, 100px, 50px, 50px);
```

And we'd get the result depicted in Figure 6.7.



Figure 6.7. Extracting a different triangle

Defining the Floor Plan

The essence of our maze script lies in our ability to create a three-dimensional perspective from a two-dimensional map. But before we can make sense of how the perspective works, we must look at the map—or, as I'll refer to it from now on, the **floor plan**.

The floor plan is a matrix that defines a grid with rows and columns. Each square in the floor plan contains a four-digit value that describes the space *around that square*—whether it has a wall or floor on each of its four sides. As we'll see in a moment, we'll use a 1 or a 0 for each of the four digits.



Understanding clip

`clip` totally confuses me—every time I use it, I have to think about how it works all over again. To help jog your memory, Figure 6.8 illustrates what the values in that clipping rectangle mean.

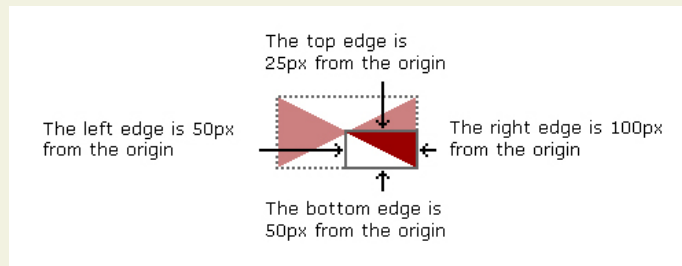


Figure 6.8. How CSS `clip` works

The main element in this example (indicated by the dotted line) is 100px wide and 50px high. The four values in the clipping rectangle are (in order): top offset, right offset, bottom offset, and left offset. Each of these values defines the offset of that edge from the main element's origin (its top-left corner).

These values are specified in the same order (top, right, bottom, left) as they are for other CSS properties, such as `border`, `padding`, and `margin`. Thinking of the word *trouble* (TRBL) should help you remember the correct order.

Figure 6.9 shows how each of these squares is constructed.

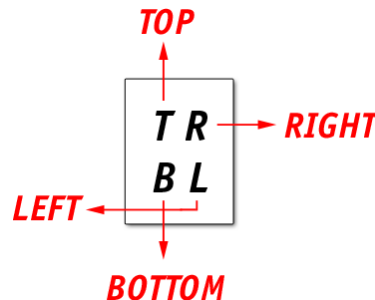


Figure 6.9. A single square describes the space around that square

Figure 6.10 shows a simple floor plan that uses four of these squares.

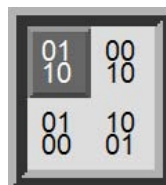


Figure 6.10. A simple floor plan example

In Figure 6.10:

- A dark gray block represents a square of solid wall.
- The borders at the edge of the diagram also represent solid wall.
- A light gray block represents a square of open floor.

For each square in the diagram:

- The digit 0 means “there’s solid wall in this direction.” Think of the number 0 as being shaped like a big brick, which means “Nope, you can’t walk here.”
- The digit 1 means “there’s open floor space in this direction.” Think of the number 1, being a positive value, as “Yes, you may walk on this square.”
- Each of the four digits in a square represents a direction when the floor plan is viewed from above. The numbers should be read left-to-right, top-to-bottom, and they should appear in the same clockwise order as CSS values: top, right, bottom, left (or, when considered from the point of view of someone within the maze: forward, right, backwards, left).

A square like the one in the top-right of Figure 6.10 therefore represents the following information:

- The four-digit number represented is 0010.
- There are solid walls above, to the right, and to the left of the square.
- There is open floor space below the square.

As you can see, the concept is rather similar to the classic Windows game, Minesweeper!

The floor plan in Figure 6.10 would be represented in JavaScript by the following matrix:

```
this.floorplan = [['0110', '0010'], ['0100', '1001']];
```

Note that these values are strings, not numbers; with numbers, leading zeros are not preserved, but in this case those leading zeros are an important part of the data.

So far, we’ve only seen very small examples of floor plan data. To make our maze really useful, we’ll want something much larger—the floor plan included in the code archive is 20 by 40 squares, and even that is comparatively small.

Just for kicks, Figure 6.11 shows what that floor plan looks like—you can refer to this plan if you get lost wandering around! As before, the light squares represent floor space and the dark squares depict solid wall, while the red cross-marks show positions where the person navigating our maze (from here on referred to as the **player**) can stand.

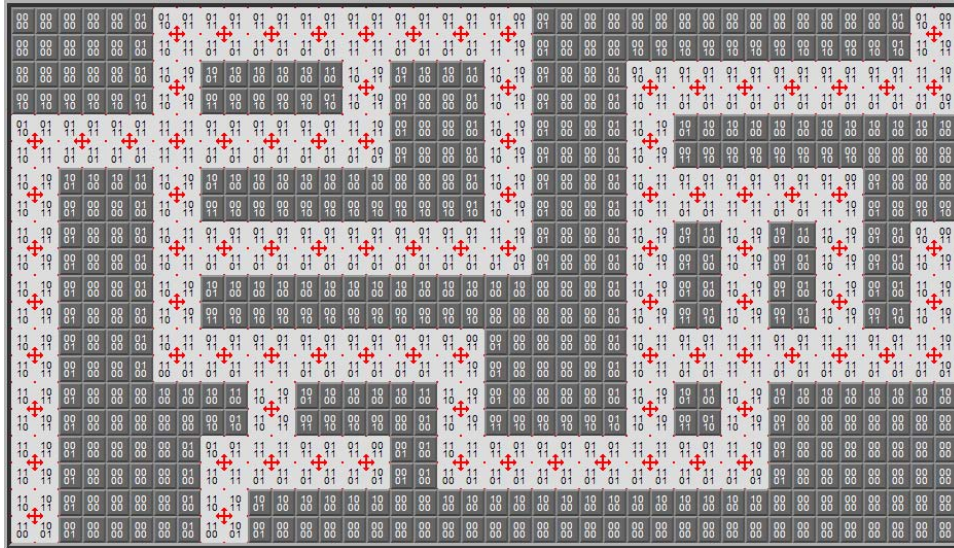


Figure 6.11. A complete maze floor plan

I don't expect you to be able to read those numbers! But later on, when we talk about the floor plan designer that goes with the game, you can look at this plan in its original context. The floor plan designer is also included in the code archive.



There are Many Ways to Skin a Cat!

There are, of course, numerous ways to approach a problem like this, each with its own pros and cons. For example, instead of binary digits, we could have used letters like WFFW to indicate wall and floor space. We could have made use of nested arrays, like `[[[0, 1, 1, 0], [0, 0, 1, 0]]]`. We could even have represented each square using only a single digit, which would certainly have made creating and modifying a floor plan easier.

The reason I chose to use four digits is because, this way, each square is able to represent *what's around it*, rather than *what the square itself is*. If we had a floor plan that used single digits, and we wanted to represent the view from the middle square, we'd need not only that square's data, but also the data from the four squares that surrounded it.

With the approach I've taken, we only need the data from the middle square to know what those surrounding squares are. Granted, we end up with some duplicate data in our floor plan. However, in terms of pure computational efficiency, the two are equivalent, and using four digits makes more sense to me as each square is much more self-contained.

Creating Perspective

Now that we understand how the floor plan works, and we've seen how to make triangles, we have all the data—and the building blocks—we need to create a 3D view.

Take a look at Figure 6.12. What this diagram shows is a breakdown of all of the elements that create the illusion of perspective in our maze. The walls on each side of the long hallway are composed of 16 columns. Each of the columns contains four inner elements which, for the rest of this chapter, we'll refer to as **bricks**. I've labeled the bricks, and highlighted them in a different color so that they're easier to distinguish. In each column, the **top brick** is highlighted as a gray rectangle; the **upper brick** is a rectangle comprising a red and blue triangle, as is the **lower brick**; and the **middle brick** is a green rectangle.

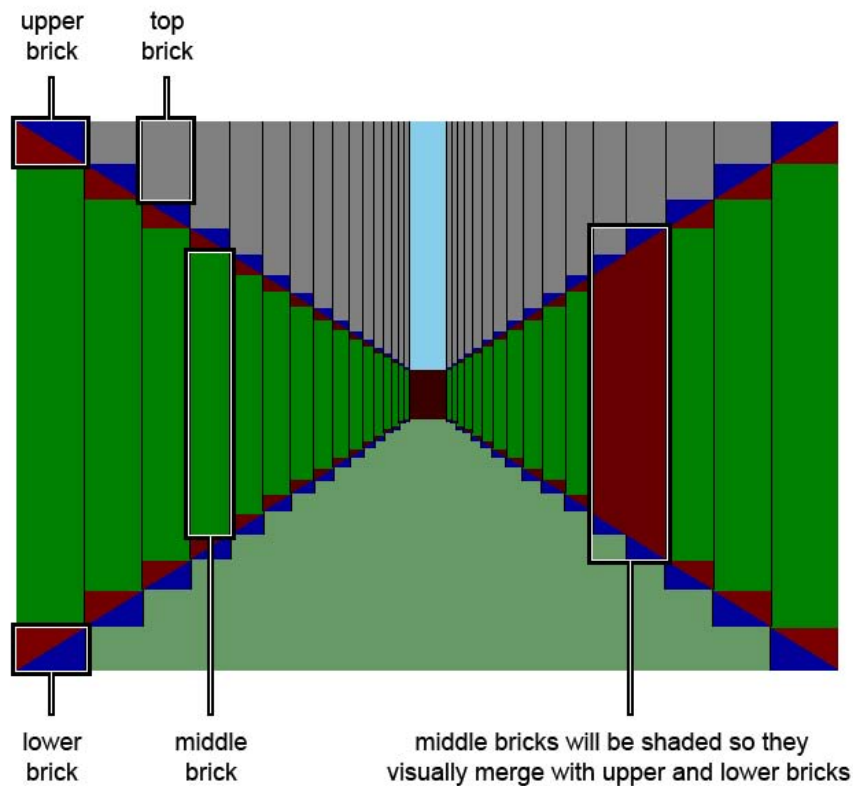


Figure 6.12. Combining the building blocks to create perspective

The upper and lower bricks are implementations of the triangles we saw earlier, clipped differently for each of the four orientations we need, thus creating diagonal lines in four directions. The red parts of these bricks will always be visible, whereas the blue parts are only blue for demonstration purposes—in practice, they'll be transparent. The top bricks will also be transparent, to expose a sky-patterned background.⁵ The middle bricks will be shaded the same dark red color as the triangles in the upper and lower bricks, so that the bricks merge together and create the appearance of part of a wall.

⁵ It isn't strictly necessary to use top bricks—we could have applied a top margin to the upper bricks—however, it was easier for me to visualize this way.



This Is Not a True Perspective!

What we're dealing with here is not actually a true perspective—it's skewed slightly so that the vanishing point is a short vertical line, rather than a point.

I originally created this maze using a true perspective with a single vanishing point, but it just didn't look right. The ceiling appeared too low relative to the distance between the walls (or the walls were too far apart, depending on how you looked at it). Changing the aspect ratio (that is, making the viewport square instead of the widescreen ratio that it has) would have made a difference, but I didn't want to do that—I wanted the game to look more cinematic!

The view is also limited as the columns get smaller, rather than stretching all the way to the vanishing point, because the resolution that we can achieve at such a distance is limited. The view ends at the point where we no longer have enough pixels to draw effectively, which restricts the maximum length of corridor we can represent. We'll talk about this issue again, along with the other limitations of this approach, towards the end of the chapter.

If you look carefully, you'll see in Figure 6.12 that each of the triangles has the same angle—it's just the size of the brick itself that's progressively reducing. This makes the illusion of perspective nice and easy to create, as we don't have any complex math to worry about. Still, it's not something that we'd want to code by hand. Let's use JavaScript to calculate the size of each brick, so that it can be generated on the fly ...

Making a Dynamic View

One of the beautiful things about using a programming language to generate complex visual patterns is that it's not necessary for us to work out every line and angle manually—we only need to worry about the math that represents the pattern.

There are times when I really wish I'd paid more attention in school math classes. But computer games were in their infancy then, and none of my teachers knew much, if anything, about them. So when I asked in class, "What use is any of this?", they didn't have a good answer!

It's just as well, then, that the math involved here is not complicated—we don't even need trigonometry, because the angles have already been determined for us. All we need to calculate is the size of the bricks and the clipping regions that are used to create our triangles; the browser's rendering engine will do the rest.

Core Methods

Let's take a look at the scripting now. We'll start with the main script, **underground.js**, which is located in the **scripts** folder of the code archive. The entire script would be too large to list in its entirety in this book; instead I've just listed the signature of each method to give you a high-level appreciation for what's going on:

underground.js (excerpt)

```

// DungeonView object constructor
function DungeonView(floorplan, start, lang, viewcallback)
{ ... };

// Create the dungeon view.
DungeonView.prototype.createDungeonView = function()
{ ... };

// Reset the dungeon view by applying all of the necessary
// default style properties.
DungeonView.prototype.resetDungeonView = function()
{ ... };

// Apply a floorplan view to the dungeon
// from a given x,y coordinate and view direction.
DungeonView.prototype.applyDungeonView = function(x, y, dir)
{ ... };

// Create the map view.
DungeonView.prototype.createMapView = function()
{ ... };

// Reset the map view.
DungeonView.prototype.resetMapView = function()
{ ... };

// Apply a position to the map view.
DungeonView.prototype.applyMapView = function()
{ ... };

// Clear the view caption.
DungeonView.prototype.clearViewCaption = function()
{ ... };

// Generate the caption for a view.
DungeonView.prototype.generateViewCaption = function(end)
{ ... };

// Shift the characters in a string by n characters to the left,
// carrying over residual characters to the end,
// so shiftCharacters('test', 2) becomes 'stte'
DungeonView.prototype.shiftCharacters = function(str, shift)
{ ... };

// Bind events to the controller form.
DungeonView.prototype.bindControllerEvents = function()
{ ... };

```

Rather than examine every method here, I'll explain the three core methods that do most of the work for our script, and leave you to fill in the gaps by following the code from the code archive yourself. Throughout this section I'll use the word **view** to mean “a 3D representation of a position on the floor plan” (that is, the player's point of view, looking north, east, south, or west).

The createDungeonView Method

The createDungeonView method takes an empty container, populates it with all the elements we need (the columns are divs, and the bricks are nested spans), and saves a matrix of references to those elements for later use:

underground.js (excerpt)

```
// Create the dungeon view.
DungeonView.prototype.createDungeonView = function()
{
    var strip = this.tools.createElement('div',
        { 'class' : 'column C' }
    );
    this.grid['C'] = this.dungeon.appendChild(strip);

    for(var k=0; k<2; k++)
    {
        // the column classid direction token is "L" or "R"
        var classid = k == 0 ? 'L' : 'R';
        for(var i=0; i<this.config.gridsize[0]; i++)
        {
            var div = this.tools.createElement('div',
                { 'class' : 'column ' + classid + ' ' + classid + i }
            );
            this.grid[classid + i] = {
                'column' : this.dungeon.appendChild(div)
            };
            for(var j=0; j<this.config.gridsize[1]; j++)
            {
                // create the main span
                var span = this.tools.createElement('span',
                    { 'class' : 'brick ' + this.bricknames[j] }
                );
                if (j == 1 || j == 3)
                {
                    var innerspan =
                        span.appendChild(this.tools.createElement('span'));
                }
                this.grid[classid + i][this.bricknames[j]] =
                    div.appendChild(span);
            }
        }
    }
}
```

```

    }
    this.resetDungeonView();
  };

```

As you can see if you scroll through the code, there isn't much more to this method: its sole responsibility is to create a group of elements, and assign `class` names to each of them so that they can be distinguished from one another. The values I've used are reasonably intuitive—`upper` identifies an upper brick, for example.

I've made use of CSS `floats` in order to line the columns up (left `floats` for a column on the left wall, and right `floats` for one on the right). To create the columns, we iterate on each side from the edge inwards (in other words, the left-most column is the first of the columns that comprise the left wall, and the right-most column is the first for the right wall).

The `resetDungeonView` Method

The `resetDungeonView` method applies style properties (`size`, `position`, `clip`, `background`, and `border-color`) to the elements that form the most basic view—that shown when our user is looking straight down a corridor that stretches the maximum distance that our script can support, as depicted in Figure 6.13.

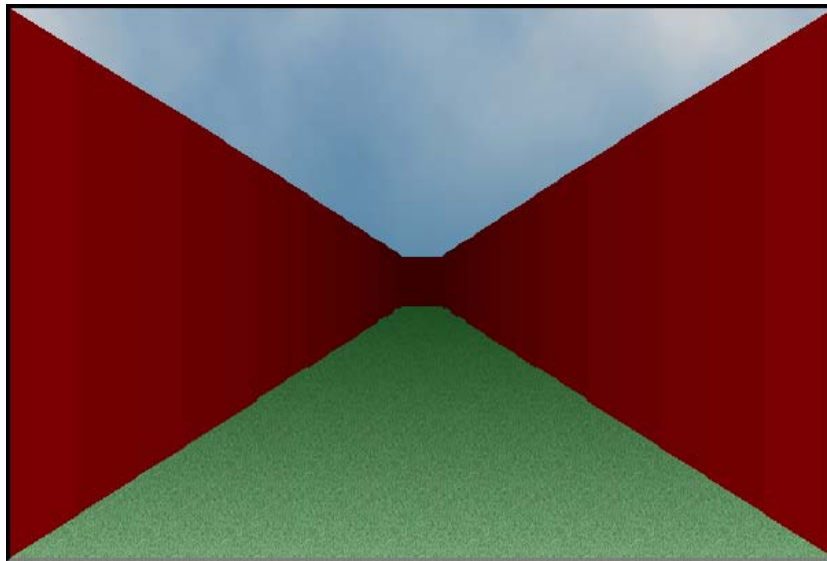


Figure 6.13. The `resetDungeonView` method rendering a basic view, without floor plan data

This method can be called whenever we need to reset the view, which we'll do at initialization, and again before applying each new view. It works by iterating through the matrix of element references we created in `createDungeonView`; it calculates the width of each column and the height of each of the bricks inside it.

To perform this calculation, we need to define some structural constants. These constants can be found in the configuration script, **config.js**, which is also in the code archive's **scripts** directory:

config.js (excerpt)

```
this.viewsize = [600, 400]; ❶  
this.gridsize = [16, 4]; ❷  
this.bricksize = [50, 31]; ❸  
this.multiplier = 0.84; ❹
```

These constants represent the following values:

- ❶ The **viewsize** represents the total width and height of the view container.
- ❷ The **gridsize** represents the number of columns from the edge of the **viewsize** to the center, and the number of bricks from top to bottom.
- ❸ The **bricksize** is the size of the upper and lower (triangle-creating) bricks.
- ❹ Finally, the **multiplier** controls the factor by which the brick size is reduced for each column as we move towards the center of the view.

Figure 6.14 shows the same perspective diagram that we saw in Figure 6.13, this time with captions indicating how each of these structural constants applies.



Working Out the Values

I'd love to say I had a clever mathematical algorithm for calculating the values I've used here (and there probably is one), but I can't. I just used trial and error until I arrived at something that looked about right. Note, however, that the values are *very closely* interrelated, so be extremely careful when adjusting them!

The choice of correct values is also dependent upon the overall performance of the script—it would be possible to create a higher resolution maze with a larger number of smaller bricks. However, that would mean we had more objects to render, which would result in lower overall performance. Even with the default values that I've set above, you need a fairly decent computer to render this maze effectively.

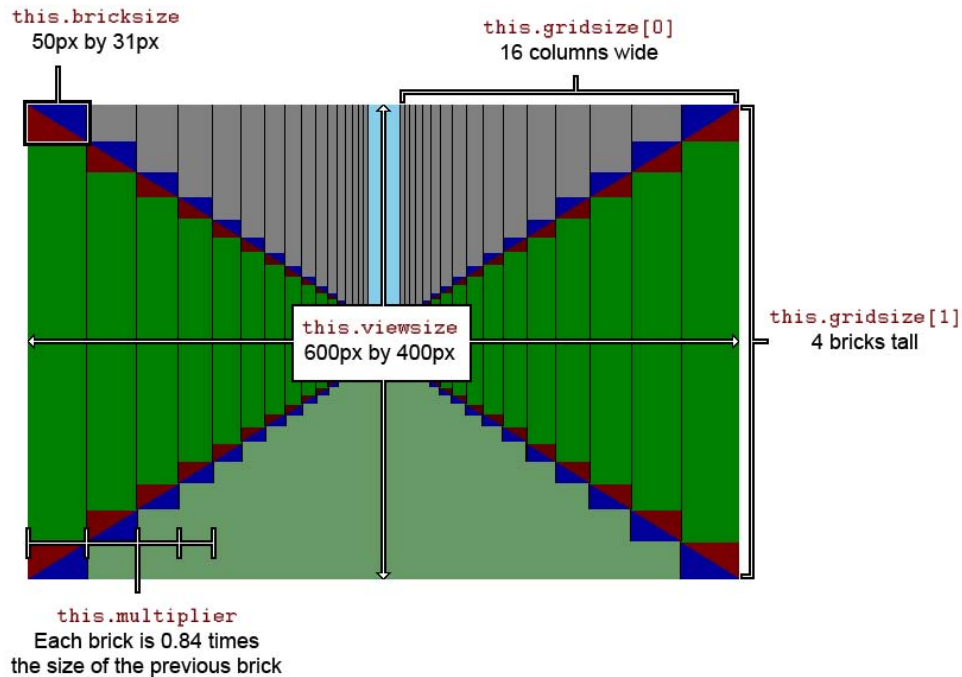


Figure 6.14. Structural constants defining the maze's perspective

If you have a look at Figure 6.14 again, you'll notice that the bricks line up perfectly—in each column, the upper brick is exactly below and to the side of the upper brick in the previous column; likewise, each lower brick lines up below and to the side of its neighbor. The `clip` and `position` values of the inner elements of those bricks decrease proportionally as the brick size decreases, while the height of the top and middle bricks changes as necessary to complete the wall.

Finally, in order to improve the appearance of perspective, we want each column to be slightly darker than the previous one. To achieve that goal, I've introduced constants that define the base color of our bricks and the darkening proportion that's applied to them. We'll define the `wallcolor` using RGB values—they're easier to work with, as the values are decimal rather than hexadecimal. We'll name the constant that controls the darkness of each column the `darkener`. Both of these constants are defined in the `config.js` file:

```
this.wallcolor = [127, 0, 0];
this.darkener = 0.95;
```

On each iteration of our code, we render a single column on each side, moving towards the center of the view; the base color is darkened by the amount specified in the `darkener` constant. I chose a dark red for the main demo (dark colors generally work best), but as Figure 6.15 shows, you can use any color you like—even pink!

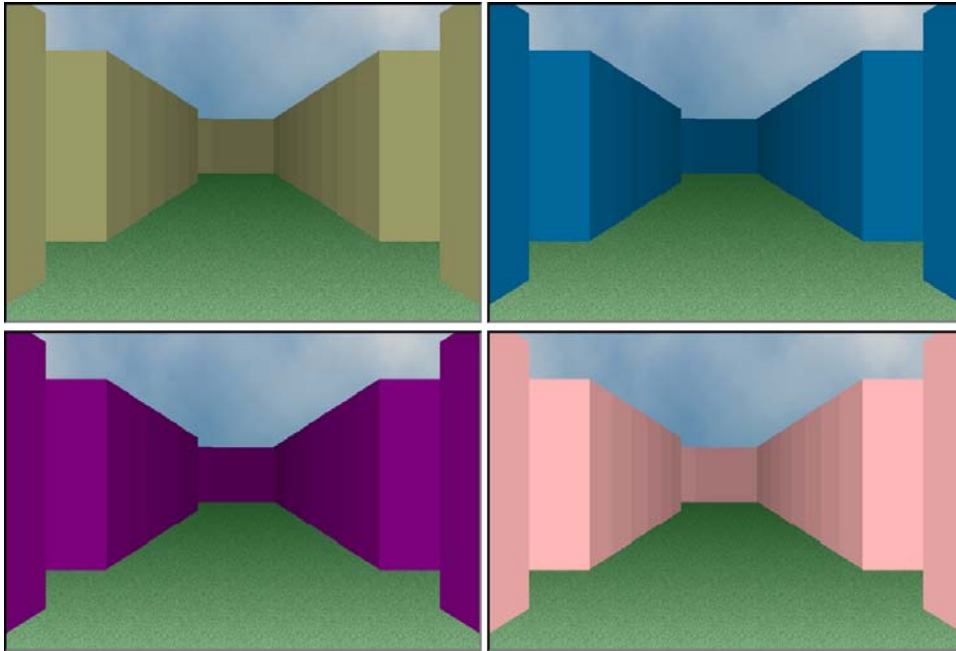


Figure 6.15. Rendering the walls of the maze with different base colors

The applyDungeonView Method

The `applyDungeonView` method applies style variations to the basic view, creating passageways off to either side of our main passage. To do this, it first compiles a matrix, stored in the variable `this.squares`, which is a subset of the complete floor plan. This matrix consists of only those floor plan squares that are necessary for us to render the player's view from the current location in the maze.

Figure 6.16 shows an excerpt of a floor plan. The green square highlights the spot where the player is currently standing, while the blue border surrounds what the player can see. It's the region inside this blue border that defines the part of the plan required to draw the view for the player.

In this example we're looking north, and each of the floor squares provides us with information about the surrounding squares. However, for any direction of movement, the player is always looking "forwards," and it's the player's view that we render. So the first thing we must do is translate the data contained within each square into data that's accurate for the direction in which the player is facing. Let me explain this with an example ...

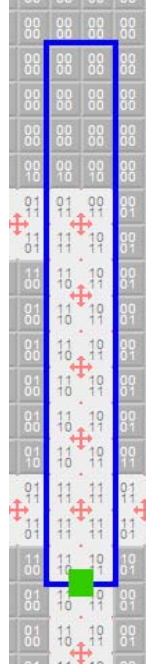


Figure 6.16. An extract from a floor plan showing the data for a single view

Remember that the digits in a square indicate the presence of wall or floor surrounding that square, in clockwise order, starting from the top. Well, we want those four digits always to indicate that information clockwise from the top, regardless of the direction in which the player is actually facing. Should we have the value 1110 when facing north, then, when the player was facing east, that same square would be represented by the value 1101. When the player faced south, the value would be 1011, as shown in Figure 6.17.

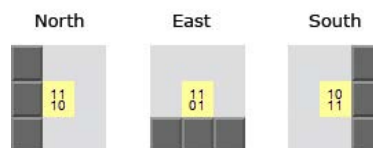


Figure 6.17. The floor plan data varying when the player looks in different directions

So, as we compile the `this.squares` matrix, we need to translate each square's value to the direction in which the player is facing. A small utility method named `shiftCharacters` performs this translation: `str` is the four-digit string, and `shift` is the number of times the square must be rotated in a counterclockwise manner when the player turns in a clockwise direction. Each turn corresponds to each of the four digits that represent that square moving to the left by one position (with the left-most digit jumping to the end of the string).

To continue with the example in Figure 6.17, if the player's view was to change from north (with floor plan data of 1110) to west (0111), the `shift` value would be 3.

The `shiftCharacters` method looks like this:

`underground.js (excerpt)`

```
DungeonView.prototype.shiftCharacters = function(str, shift)
{
  var saved = str.substr(0, shift);
  str = str.substring(shift);
  str += saved;
  return str;
};
```

Once we have the data we need, we can iterate through it and create the actual view. This is where things get rather tricky.

First of all, we need to iterate forwards through the squares, starting from the player's current location. With each iteration, we test the first digit of each square (which tells us what's in front of it) until we find the end wall. The end wall marks the limit of what the player can see—every column from that point onwards should be assigned the same height and color. These columns will create the illusion of a facing wall, as shown in Figure 6.18.

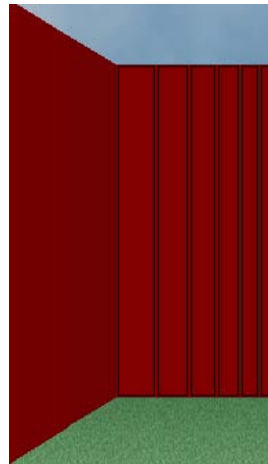


Figure 6.18. Columns combining to form a facing wall

Once we know the limit of the player's view, we iterate from that point *backwards* through the floor plan data towards the player's location, looking for adjoining passageways. We need to iterate backwards because the height of a passageway's facing wall is the height of the furthest column that defines it.

To illustrate, Figure 6.19 shows another excerpt from the perspective diagram, this time with lines and shading overlaid to show a corridor with a passageway off to the left.

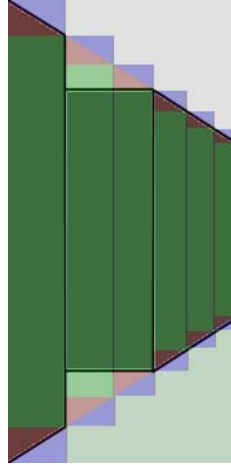


Figure 6.19. Constructing perspective to create a passageway off to the left

If we want those second and third columns to create that passage to the left, we need to remove the upper and lower bricks from those columns, leaving only the middle bricks, which then must be resized as necessary. But our passage is two columns across, and it's the furthest column (or what we might call the **corner column**) that determines the height of the wall—not the nearest. So we need to modify that corner column first, so that we know how tall to make the adjacent columns.

Iterating forwards would require us to jump two steps ahead to find the corner, then move one square back to make a further adjustment. And that's why we iterate backwards, rather than forwards. (I told you it was tricky!)

When we create those passageways, we also lighten the facing walls slightly, to improve the visual appearance and make the wall look more realistic. As we did when we darkened the walls, we use a single constant value (I've called it the `lightener`) to determine the amount of lightening required:

```
this.lightener = 1.25;
```

As with the height value, the lightening is applied to the corner column first, then copied onto the nearer column (for the same reasons). And once again, as with all of the constants used in this script, I have no magic formula to share for how these values were obtained—they're just what looked right after trial and error.

Figure 6.20 shows the same view excerpt again—this time without the exposed construction—looking as it does in the final game.

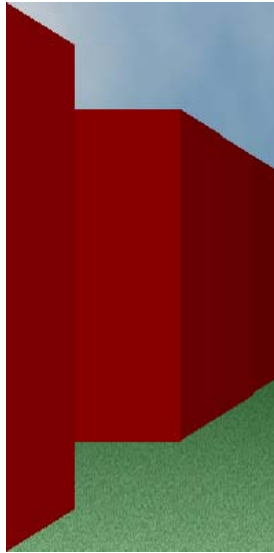


Figure 6.20. A passageway off to the left

Applying the Finishing Touches

Now, I hope, you should have a fairly concrete sense of how the script generates perspective views, with walls and passages created as necessary. From the diagrams we've seen so far, you can understand that any given view is simply a combination of rectangles and triangles.

One final touch we'll need to make is to shift the entire view up inside the container in order to raise the horizon slightly. This is just another visual tweak that I included because I think it produces a better-looking and more realistic result, as Figure 6.21 shows.

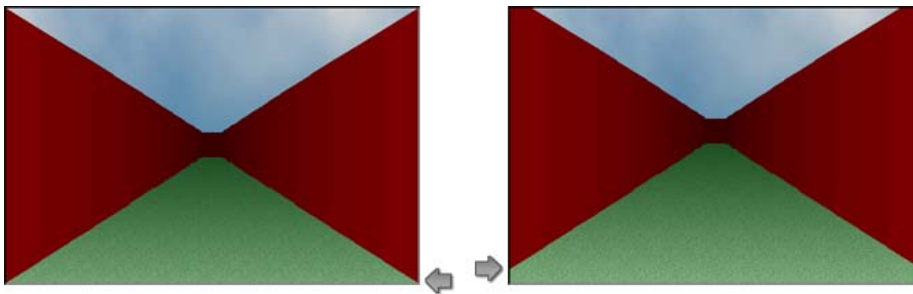


Figure 6.21. Adding a slight horizon shift to the overall view

You'll notice I've used images for the sky and floor patterns. These images provide some texture to add to the realism of my maze; they also contain a slight gradient, growing darker as they approach the horizon, which again reinforces the sense of perspective.

The end result is not perfect, though: unavoidable rounding errors occur in the final output figures, and these errors give rise to an occasional discrepancy of one or two pixels between adjacent

columns. The shading computation is not exact either—sometimes, on close walls, you can see a slight color difference between two columns that should be exactly the same.

All things considered, however, what we’ve created here is a reasonably convincing 3D maze.

Limitations of This Approach

The approach we’ve taken to build this maze imposes some limitations on the design of a maze floor plan, thus restricting the kind of layout we can draw:

- Corridors must always be two squares wide—we can’t create wider spaces because we don’t have the pieces with which to draw them.
- No single corridor can be longer than 16 squares, as this is the maximum number of pairs of columns that we can draw.
- Walls must also consist of an even number of squares—every block must comprise a block of at least two squares by two squares.

It may help to think of four squares on the floor plan as one single square; those smaller squares only exist so that we have more elements to apply progressive shading to, and hence achieve a better-looking and more realistic 3D view.

Creating the Map View

To the right of the maze view, we’ll add a map that shows the floor plan in the player’s immediate location. I originally added this feature to display a top-down view of the same view that the player can actually see ... but then I realized—what’s the point of such a map, if it provides no extra advantage?

Instead, we’ll add a map that shows a little more of the surrounding area, as an aid to orientation. In the view shown in Figure 6.22, you can see that the player can only move a short distance forwards before reaching a wall, but the map to the right shows further corridors beyond that wall.

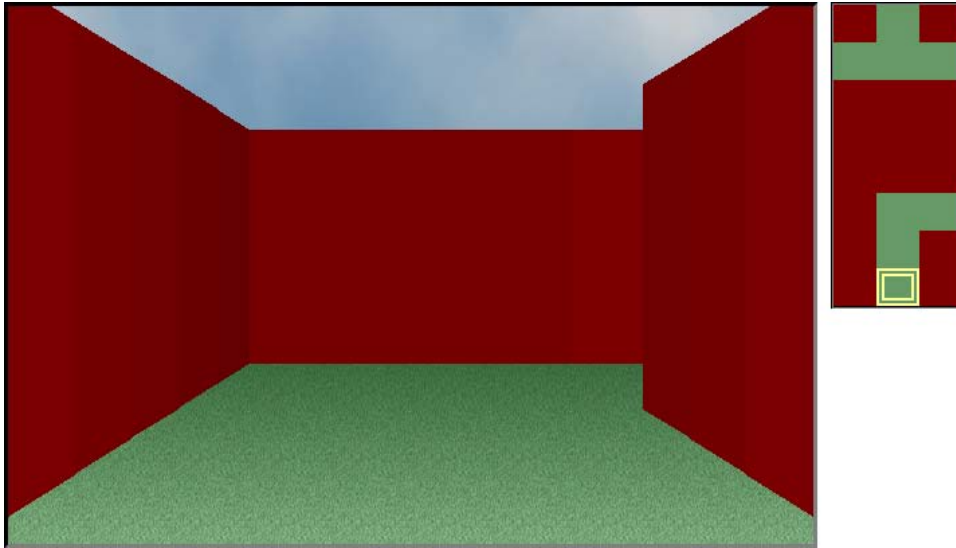


Figure 6.22. Showing extra information in the player's map

The construction of the map itself is very simple—it's just a bunch of spans floated in a container. I've applied a solid background where there's wall, and transparency where there's floor. This allows the green background of the container to show through, as Figure 6.23 reveals.

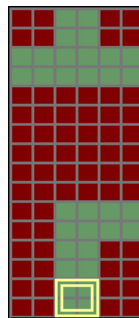


Figure 6.23. How the player's map is constructed

Generating the map is equally simple, since it's just a two-dimensional representation of data that is itself a 2D matrix.

Remember that when we generated the maze view, we created a matrix called `this.squares`. This matrix contained as much of the floor plan as was required to generate the current view, with the data transposed so that it represented a forwards view for the player. Well, we can use that same data matrix to generate this 2D map.

To create the map, we begin by coloring every square (using the base `wallcolor` property). Then we iterate through the matrix of squares, and apply transparency to every square in the map that represents open floor space—including the space directly beneath the spot where the player is standing. The `applyMapView` method in the file `underground.js` takes care of this for us:

underground.js (excerpt)

```

DungeonView.prototype.applyMapView = function()
{
  this.resetMapView();
  for(var i=0; i<this.squares.L.length; i++)
  {
    var n = this.mapsquares.length - 2 - i;
    if(this.mapsquares[n])
    {
      if(this.squares.L[i].charAt(3) == '1')
      {
        this.mapsquares[n][0].style.background = 'transparent';
        this.mapsquares[n][1].style.background = 'transparent';
        if(i == 0)
        {
          this.mapsquares[n+1][0].style.background = 'transparent';
          this.mapsquares[n+1][1].style.background = 'transparent';
        }
      }

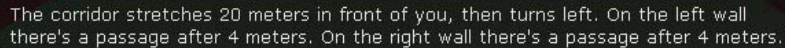
      if(this.squares.R[i].charAt(1) == '1')
      {
        this.mapsquares[n][4].style.background = 'transparent';
        this.mapsquares[n][5].style.background = 'transparent';
        if(i == 0)
        {
          this.mapsquares[n+1][4].style.background = 'transparent';
          this.mapsquares[n+1][5].style.background = 'transparent';
        }
      }

      if(this.squares.L[i].charAt(1) == '1')
      {
        this.mapsquares[n][2].style.background = 'transparent';
        this.mapsquares[n][3].style.background = 'transparent';
        if(i == 0)
        {
          this.mapsquares[n+1][2].style.background = 'transparent';
          this.mapsquares[n+1][3].style.background = 'transparent';
        }
      }
    }
  }
};

```

Adding Captions

One of the things that excites me most about web programming is its potential for improving accessibility. Although we're making a visual game here, we have data in a format that can easily be translated into other kinds of output, such as plain text. We can use the same information that we used for making the map to generate a live text description of each maze view, of the kind shown in Figure 6.24.



The corridor stretches 20 meters in front of you, then turns left. On the left wall there's a passage after 4 meters. On the right wall there's a passage after 4 meters.

Figure 6.24. A generated caption that describes a maze view

Not only does captioning potentially aid comprehension for players who have a cognitive or visual disability, it also extends the basic game play to people who are completely blind—suddenly we can navigate around the maze without any visuals at all! Admittedly, and unfortunately, the game will be much harder to play like this—not just because you have to hold orientation information in your head, but because you don't have the map to refer to in order to gain clues about what's behind the next wall.

Still, it's a start. Try viewing the game with CSS disabled, and you'll get a basic sense of the experience of what it would be like to play the game if you were blind. I've also confirmed that the game is playable in the JAWS 8 screen reader.

Generating the core data for the captions is straightforward—we simply need to know how many passageways there are to the left and right, and how far away they are. We can work this out by:

- iterating once again through the `this.squares` matrix
- building arrays to store the index of each opening

These openings will be converted to a perceived distance. As we navigate our maze, one square looks to be roughly two meters in length, so we'll adopt this as the scale for our map. We can stop iterating once we reach the end of the player's view—we've created an `end` variable in the `applyDungeonView` method, which is the index of `this.squares` at the point that the view ends. Therefore, we can simply pass this value to the `generateViewCaption` method when we call it.

In the code, I've used `len` to represent the total length of the corridor in front, and arrays called `passages.left` and `passages.right` to store the distance of each passage from the player. The result of our iterations might produce data like this:

```
var len = 16;
var passages = {
  'left' : [8, 16],
  'right' : [4]
};
```

This looks simple enough to interpret, right? Well, yes ... however, turning this data structure into coherent English is still a little tricky. The *basic* conversion is easy. Using the data we have, we can describe the view in coarse terms:

“The corridor stretches 16 meters in front of you. To the left there are passages after 8 meters and 16 meters. To the right there are passages after 4 meters.”

However, this language is fairly obtuse. For one thing, we wouldn’t want to say “there are passages” if there was only one. Instead, we’d want to say “there’s a passage.” Additionally, the last passage to the left is at the far end, so it would be nicer to describe that by saying “The corridor stretches 16 meters in front of you, then turns left.”

We also need to deal with exceptions. For example, if the player is standing directly in front of a wall, we don’t want to say “... stretches 0 meters in front ...” Likewise, if the player has just turned right into a passage, we don’t want to say “to the right there’s a passage after 0 meters.”

To cater for all these exceptions, the script accepts a dictionary of sentence fragments with replacement tokens, which are then compiled and parsed as necessary, in order to obtain a result that approaches decent prose. If you have a look in `init.js`, you’ll notice that the `DungeonView` object is instantiated with this data as an argument. Each of the language properties is a sentence fragment with replacement tokens; for example, `%dir` is a direction token that will be replaced with the word for “left” or “right,” as applicable.

I’d encourage you now to scroll through the `generateViewCaption` method in `underground.js`, and read the comments there that explain each situation. As it is, there’s still room for improvement, but this is one of those things that you could refine to the n^{th} degree, and it would still never be perfect.⁶ That said, I believe that the end result is fairly good—the captions are verbose enough to get the information across, they’re succinct enough not to be arduous to read, and they flow well enough that they don’t sound too much like they were generated by a machine (even though they were!).

Designing a Floor Plan

In the code archive for this book, you’ll find a **floor plan designer**, which is a separate JavaScript application that generates the `floorplan` matrix used by this game. It’s a table of squares, and you

⁶ Read more about the problems associated with constructing natural-sounding sentences in English in the Wikipedia entry on natural language processing at http://en.wikipedia.org/wiki/Natural_language_processing/.

can click a square to toggle it between floor and wall. The script will work out the numbers for each square that relate to that view, using the TRBL syntax I introduced earlier in the chapter to denote whether a square has wall or floor on each of its four sides.

Hovering over a square in the floor plan designer will also display a tooltip containing the `x,y` position of that square in the grid. This information is useful for defining a start position (the first two values of the `start` array in `init.js`).

To use the floor plan designer, first create your plan by clicking on the squares. When you're happy with your maze, click the **Generate output matrix** button and a `floorplan` matrix will be generated for you. You can then copy and paste this data directly into your `init.js` file—the next time you run the maze application, your new floor plan data will be passed to the script.

Alternatively, you can begin your floor plan editing session by pasting existing floor plan data into the `textarea` field. Click **Display input matrix**, and the floor plan designer will display the map representation of the data that you pasted into the field, which you can then edit further as required. Try pasting in the original `floorplan` matrix from `init.js`, and you'll see the plan that I showed you near the start of this chapter, in all its glory!

Simple as it is, without this tool, making the maze floor plan would be a very painful process! In fact, I created this tool before I wrote the main script.

Further Developments

Before we close this chapter, I'd like to take a couple of moments to discuss some general possibilities for further development of the maze. More specifically, we'll look at the **callback** facility that's available for hooking additional code into each view change.

Using the Callback

Have a look in `init.js` and you'll notice that, in addition to the floor plan, start position, and language parameters, there's an optional fourth argument specifying a `viewchange` callback function. This function will be called every time a new view is drawn, and can be used to add logic to the game.

The `viewchange` function referred to in this example can be found in the script called `demogame.js`, which is located in the **addons** directory of the code archive. This script and its associated style sheet are both included in `underground.html`, at the very end of the `head` section (after the core style sheets and scripts).

As you'll see, the callback accepts the following arguments:

- x** the current `x` position of the player
- y** the current `y` position of the player

dir the direction that the player is currently facing

inst a reference to this instance of the `DungeonView` object

By defining conditions based on the first three arguments, you could add logic that applies only at specific locations in the maze. And because the callback function will always be called when the player begins navigating the maze at the start position, you could also use the callback function for initialization code. For example, a flag could be set to indicate that a location-specific action has occurred, so that it occurs only once.

The fourth argument, *inst*, is a reference to this instance of `DungeonView`, and can be used for tasks like adding a new element to the view (such as objects for the player to find), or modifying the configuration properties (in order to change the wall color in certain areas of the maze).

In the demo game example, I've made use of the callback function at one specific position in the floor plan—at this point in the maze you can see a simple object in front of you, and at another position you're standing directly above that object (that is, picking it up). That's all there is to the demo game—there's nothing ground-breaking—but at least it adds an end purpose to an otherwise aimless meander through the maze! It should also serve to illustrate the principle of extending the maze, and will hopefully inspire you to try something more ambitious and creative.

At sitepoint.com, you can find a more sophisticated example in which a hidden surprise is located within a larger maze, and your mission is to find it.⁷

Blue-sky Possibilities

It would be quite simple to use Ajax to relay a player's position to a server—other players could read that data, thus facilitating the creation of an online multiplayer environment. It should also be possible to implement a server-side program that generates floor plan data and sends it back to the game, effectively creating multiple “levels” in the maze. Taking this idea one step further, players could potentially receive and transmit floor plan data between themselves, thereby allowing individuals to host maze levels.

However, it would be quite tricky to represent other players in the view—we would need a graphic for every additional player, as well as versions of that graphic at each of eight different distances, facing in four directions. Short of generating the players as simple shapes, there's no pure-CSS way to create these graphics. They would have to be a collection of specially drawn images, and I don't have the artistry to design those characters!

But if you do, be my guest. If you had those images, adding them to the game would be most simply achieved with absolutely positioned overlays—placing the image so that its center is in the center of the maze. Then, for each view, it would be a case of working out which was the correct image to

⁷ Visit <http://maze.sitepoint.com/> to play this game.

show, based on the locations of that player relative to the main player. This might also be quite tricky, especially when you had three or more players sharing the same corridor, but I have no doubt that it's doable.

Who knows—maybe you could add combat too!

Summary

In this chapter, we took the languages of CSS and JavaScript well beyond the tasks for which they were intended—the presentation and basic behavior of HTML documents—and used them to create an interactive 3D maze.

First, we looked at the basic principles by which triangles can be displayed using only CSS. We then extended that concept to render a perspective view, creating the illusion of three dimensions. Next, we established a convention for specifying floor plan data, and for dynamically translating that data into a perspective view. By adding listeners for user events, we successfully created an interactive maze that can be completely customized and extended. To top things off, we added some usability aids, such as a top-down map, and accessibility aids including keyboard navigation and captions.

While I haven't delved into the details of every method that comprises the game script (there are plenty of comments, so I'll leave that for you to pursue in your own time), I hope this chapter has convinced you to look at JavaScript in a new light. The possibilities really are only limited by your imagination!

What's Next?

If you've enjoyed these chapters from *The Art & Science of JavaScript*, why not order yourself a copy?

You've only seen two of the projects covered in this ground-breaking book -- if you enjoyed the sample, then the rest of the book will blow your mind!

The Art & Science of JavaScript will teach you how to:

- Create a slick Google Maps and Flickr mashup.
- Give your site some extra personality with client-side badges.
- Write better code faster using metaprogramming techniques.
- Create stunning vector graphics with the canvas element.
- Become a debugging expert using pro-level Firebug tricks.

The complete JavaScript, HTML and CSS code used to create each of the projects is available for download, and is guaranteed to be best practice and ready to use.

If you're looking for a JavaScript book that will inspire you to build the next BIG web application, then this is the book for you. With plenty of screenshots and in-depth explanations, *The Art & Science of JavaScript* will change the way you approach JavaScript forever!

Take your JavaScript to the next level now. Have *The Art & Science of JavaScript* ordered direct to your door from us, the publisher.

[Order the full version now!](#)

Index

A

- aIdx variable, 12
- Ajax
 - client-side badges, 53
- alpha variable, 12
- anchors
 - sorting tables, 8
- anonymous functions, 157
- Application Programming Interfaces (API)
 - (*see also* Flickr API; Google Maps API)
 - canvas element, 77
 - defined, 218
 - del.icio.us, 51
 - using dynamic functions, 172
- arcs
 - drawing with canvas element, 91
 - pie chart segments, 102
- arguments
 - default, 164
- arguments array
 - about, 157
- arrow keys
 - alternatives to, 28
- aspect-oriented programming, 171
- assertions
 - Firebug, 136
- auto-completion
 - Firebug, 137
- avatar images
 - (*see also* thumbnails)
 - badges, 47

B

- backup
 - servers, 72

- badges
 - (*see also* client-side badges; server-side badges)
 - about, 46–53
 - defined, 45
- behavior classes
 - defining, 176
- Bézier curves
 - drawing with canvas element, 93
- blacklisting
 - Firebug, 127
- blogs
 - badges, 51
- bookmarklets
 - Firebug, 137
- branching
 - functions, 168
- breakpoints (*see* conditional breakpoints)
- browsers
 - onmousedown event handler, 32
 - ternary operators, 13
 - XPath support, 169
- bubble sort
 - tables, 14
- built-in functions
 - using, 165
- built-in objects
 - adding methods to, 166

C

- callbacks
 - creating mazes with CSS and JavaScript, 214
- callee property
 - storing values between function calls, 159
- canvas element, 75–119
 - about, 76
 - creating pie charts, 98–115
 - creating vector graphics, 76–97

- Internet Explore browser, 115–118
- captions
 - adding to maze view, 212–213
- cellIndex variable, 12
- centering
 - maps, 244
- charts (*see* pie charts)
- circles
 - drawing with canvas element, 91
- class names
 - tables, 11
- classes
 - (*see also* behavior classes)
 - in JavaScript, 153
 - Prototype 1.6, 180
- clearRect method
 - Opera browser, 113
- client-side badges, 45–73
 - about badges, 46–53
 - Ajax and JSON, 53–59
 - scripting, 59–72
 - sever backup, 72
- client-side scripting
 - badges, 48, 51
- clip code, 194
- closures
 - about, 159
 - clickable thumbnails, 239
 - partial function application, 161
 - tables, 10
- color
 - creating in canvas element, 100
- columns
 - in HTML tables, 1
 - sorting, 7–24
- conditional breakpoints
 - Firebug, 141
- configuration variables
 - badges, 63–64
- connections
 - fallbacks, 58–59

- Console
 - Firefox, 134
- Console view
 - Firebug, 128
- constructor functions
 - (*see also* dynamic constructors)
 - about, 153
 - new operator, 154
- controls (*see* map controls)
- cross-domain JSON
 - on demand, 230
 - same-origin restriction, 225
- cross-linking
 - within Firebug, 133
- cross-site scripting (XSS) attacks
 - cross-domain JSON, 226
 - eval function, 164
- CSS
 - building mazes, 189–216
 - info windows, 244
 - stretching dimensions of canvas objects, 81
- CSS styles
 - out-of-the-box badges, 50
- CSS view
 - Firebug, 130
- cubic curves
 - drawing using canvas element, 95
- curves
 - drawing with canvas element, 91

D

- data (*see* profile data)
- default arguments
 - creating functions, 164
- del.icio.us social bookmarking site
 - badge example, 51
- dimensions
 - canvas elements, 81
 - pie chart segments, 102
- Document Object Model (DOM)
 - about, 4

- DOM Builder, 173
- DOM methods
 - info windows, 242
- DOM subview
 - Firebug, 140
- DOM tree
 - accessing table elements, 4
- DOM view
 - Firebug, 131
- domain-specific languages (DSL), 183
- dragging
 - columns, 24–44
- dynamic class creation
 - JavaScript equivalent, 176
- dynamic constructors
 - creating, 176
- dynamic functions
 - APIs, 172
- dynamic view
 - creating mazes with CSS and JavaScript, 198–209

E

- elements
 - inspecting with Firebug, 133
- error console
 - Firefox and Firebug, 129
- errors
 - Firebug, 138
- eval function
 - cross-site scripting (XSS), 164
 - security, 224
- event objects
 - badges, 67
- events
 - assigning, 10
 - handling, 9, 29
- excanvas.js file, 115
- exceptions
 - creating mazes with CSS and JavaScript, 213

- executing
 - with Console, 134
- ExplorerCanvas, 115
- extensions
 - canvas element and HTML, 77
 - Firefox, 147

F

- fallbacks
 - badges, 58–59
- feeds (*see* Flickr feeds)
- Firebug, 121–147
 - components of, 127–133
 - enabling and disabling, 127
 - getting started, 123
 - installing, 122
 - using, 133–144
- Firebug Lite
 - about, 145
- Firefox browser
 - extensions, 147
 - Firebug addon, 150
 - rendering canvas element, 104
- Firefox error console, 129
- Flash
 - animation, 75
- Flickr API
 - data ownership policy, 222
 - geotagging photos, 221
 - origin of, 218
 - web resources for, 249
- Flickr feeds
 - RSS and Atom, 223
- floor plans
 - creating mazes with CSS and JavaScript, 193–196, 213–214
- for-in loop, 152
- function calls
 - storing between function calls, 159
- function factories
 - clickable thumbnails, 239

functions

(*see also* anonymous functions; built-in functions; constructor functions; dynamic functions; eval function; global functions; lazy function evaluation; partial function application; self-executing function patterns; self-optimizing functions; wrapper functions)

creating, 155

default arguments, 164–165

as objects, 155

G

geotagging

photos, 221

getElementById

accessing table elements, 4

global functions

closure, 160

global variables, 236

globalCompositeOperation property

Opera browser, 113

values, 106

Google maps

creating from Flickr photos, 233–249

Google Maps API

info windows, 241

origin of, 218

web resources for, 249

GPS

geotagging photos, 221

graphics

(*see also* avatar images; Flickr API; photos; thumbnails; vector graphics)

badges, 47

graphics context

canvas API, 77

H

handlebars

Bézier curves, 94

hCalendar microformat, 184

hCard specification

implied n optimization, 184

heading states

tables, 19

HTML extensions

canvas element, 77

HTML view

Firebug, 129

HTTP requests

badges, 47

I

ice cream cone

drawing with canvas element, 93

implied n optimization

hCard specification, 184

info windows

displaying, 241

Google Maps API, 220

inheritance

property chain, 154

initializing

pages in canvas element, 103

innerHTML property, 9

inspect command

Firebug, 133

installing

Firebug, 122

instanceof

operator, 153

Internet Explorer browser

canvas element, 115–118

onmousedown event handler, 32

introspection

objects, 152

J

JavaScript Object Notation (JSON)

- badge example, 51
- badges, 54, 67
- creating Google maps, 233–249
- handling photos, 223–233

JSON-P (JSON with Padding) (*see* cross-domain JSON)**K**keys (*see* arrow keys)**L**

Layout subview

- Firebug, 139

lazy function evaluation, 169

libraries

- (*see also* Prototype library)
- self-executing function patterns, 163

log messages

- retaining type in, 135

logging

- with Console, 134

M

map controls

- adding, 238

map view

- of mazes, 209–211

maps

- recentering, 244

mashups

- (*see also* Flickr API; Google Maps API)
- defined, 218
- origin of, 219

mazes

- building with CSS and JavaScript, 189–216

metaprogramming, 149–188

- defined, 149
- overview, 150–164

- techniques, 164–187

methods

- (*see also* private methods; public methods)
- adding to built-in objects, 166
- mixing in, 178
- stealing for other objects, 158

microformats, 184

Microsoft debugging tools, 146

mixing in

- methods, 178

module pattern

- about, 163
- scripting, 61

monitoring

- with Console, 134

mouse

- dragging columns without, 37

N

names

- classes, 11

Net view

- Firebug, 132

new operator

- constructor functions, 154

nIdx variable, 12

nodes

- in DOM tree, 4

numeric variable, 12

O

object literal notation, 39

object orientation

- simulating traditional, 178

objects

- (*see also* built-in objects)
- functions as, 155
- properties, 151
- stealing methods from, 158

- offsets
 - in canvas element, 108
- onkeydown event
 - versus onkeyup event, 37
- onkeyup event
 - versus onkeydown event, 37
- onmousedown event handler, 31
- onmousemove event handler, 33
- onmouseup event handler, 36
- Opera browser
 - clearRect and globalCompositeOperation, 113
- optimization (*see* implied n optimization; performance)
- out-of-the-box badges, 48

P

- pages
 - initializing in canvas element, 103
- parsing
 - tables, 12
- partial function application
 - closures, 161
- paths
 - shapes, 87
- patterns (*see* module pattern; self-executing function patterns)
- performance
 - (*see also* implied n optimization)
 - badges, 47
 - rendering mazes, 202
 - tuning with Firebug, 143
- perspective
 - in floor plans, 196–198
- photos
 - (*see also* avatar images; Flickr API; thumbnails)
 - geotagging, 221
 - in Google Maps, 233
- pie charts
 - creating with canvas, 98–115

- private methods
 - badges, 67–72
- profile data
 - using, 144
- progressive enhancement
 - badges, 59
- properties
 - assigning functions to, 156
 - objects, 151
 - storing between function calls, 159
- prototype chains
 - inheritance, 154
- Prototype library, 178
- prototype property
 - constructor functions, 154
 - using, 8
- public methods
 - badges, 64–67

Q

- quadratic Bézier curves
 - drawing with canvas element, 94

R

- rapid application development
 - Firebug, 133
- recentering
 - maps, 244
- rectangles
 - creating with canvas element, 80
- rendering
 - canvas elements in FireFox browser, 104
 - performance, 202
- rows
 - accessing with tables, 4
- rows variable, 12

S

- same-origin restriction, 224

- scope
 - handling, 9
- Script view
 - Firebug, 131
- search engine optimization (SEO)
 - badges, 50
- searching
 - in Firebug, 134
- security
 - cross-domain JSON, 226
 - eval function, 164, 224
 - same-origin restriction, 225
- self-executing function patterns
 - about, 161
- self-optimizing functions
 - creating, 168
- server backup
 - badges, 72
- server-side badges
 - about, 50
- shadows
 - creating in canvas element, 104–109
- shapes
 - creating with canvas element, 79
- states (*see* heading states)
- storing
 - properties between function calls, 159
- stroke operations
 - paths, 91
 - rectangles, 86
- strokes
 - re-stroking lines, 103
- Style subview
 - Firebug, 138

T

- table elements
 - accessing with `getElementById`, 4
 - accessing with `getElementsByName`, 6
- tables, 1–44
 - dragging columns, 24–44

- sorting columns, 7–24
 - structure of, 1–7
- tbody element, 4
- ternary operators
 - browsers, 13
- th variable, 12
- threads
 - tables, 9
- thumbnails
 - badges, 47
 - clickable, 239
 - displaying, 226
 - highlighting current, 240
- trackbacks
 - blogs and badges, 51
- triangles
 - creating with CSS and JavaScript, 190
- typeof operator, 152
- types
 - detecting, 152

U

- updating
 - charts dynamically, 109
- user experience
 - badges, 48

V

- validity
 - canvas element, 77
- variables
 - (*see also* global variables)
 - configuration, 63
- vector graphics, 75–119
 - canvas element, 76–97
 - pie charts, 98–115
- Vector Markup Language (VML)
 - and canvas element, 115
- views
 - (*see also* dynamic view; map view)

- Firebug, 128
- switching, 132

W

- Watch subview
 - Firebug, 142
- whitelisting
 - Firebug, 127
- widgets
 - (*see also* Flickr API; Google Maps API)
 - defined, 218
- wrapper functions, 171

X

- XMLHttpRequest objects
 - Ajax requests, 53
 - fallbacks, 58
 - same-origin restriction, 224
- XPath support
 - browsers, 169

Y

- Yahoo! User Interface (YUI) library
 - self-executing function patterns, 163
- YSlow
 - about, 146