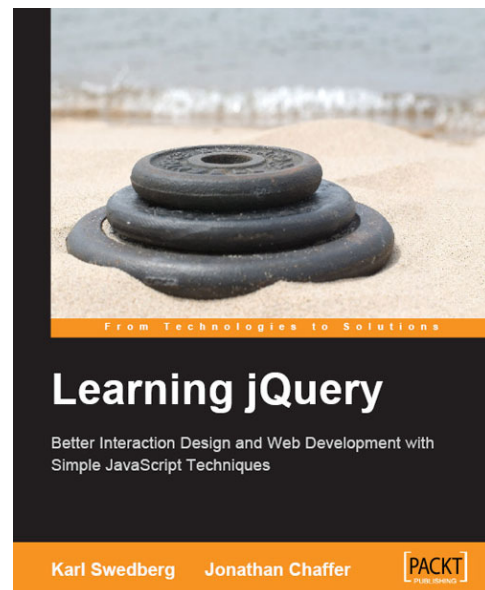




Learning jQuery:

Better Interaction Design and Web Development with Simple JavaScript Techniques

Karl Swedberg
Jonathan Chaffer



Chapter No. 7

"Table Manipulation"

In this package, you will find:

A Biography of the authors of the book

A preview chapter from the book, Chapter No. 7 "Table Manipulation"

A synopsis of the book's content

Information on where to buy this book

About the Authors

Jonathan Chaffer is the Chief Technology Officer of Structure Interactive, an interactive agency located in Grand Rapids, Michigan. There he oversees web development projects using a wide range of technologies, and continues to collaborate on day-to-day programming tasks as well.

In the open-source community, Jonathan has been very active in the Drupal CMS project, which has adopted jQuery as its JavaScript framework of choice. He is the creator of the Content Construction Kit, a popular module for managing structured content on Drupal sites. He is responsible for major overhauls of Drupal's menu system and developer API reference.

Jonathan lives in Grand Rapids with his wife, Jennifer.

Karl Swedberg is a web developer at Structure Interactive in Grand Rapids, Michigan, where he spends much of his time implementing design with a focus on web standards semantic HTML, well-mannered CSS, and unobtrusive JavaScript.

Before his current love affair with web development, Karl worked as a copy editor, a high-school English teacher, and a coffee house owner. His fascination with technology began in the early 1990s when he worked at Microsoft in Redmond, Washington, and it has continued unabated ever since.

Karl's other obsessions include photography, karate, English grammar, and fatherhood. He lives in Grand Rapids with his wife, Sara, and his two children, Benjamin and Lucia

For More Information: www.PacktPub.com/jquery/book
--

Learning jQuery: Better Interaction Design and Web Development with Simple Javascript Techniques

jQuery is a powerful JavaScript library that can enhance your websites regardless of your background.

Created by *John Resig*, jQuery is an open-source project with a dedicated core team of top-notch JavaScript developers. It provides a wide range of features, an easy-to learn syntax, and robust cross-platform compatibility in a single compact file. What's more, over a hundred plug-ins have been developed to extend jQuery's functionality, making it an essential tool for nearly every client-side scripting occasion.

Learning jQuery provides a gentle introduction to jQuery concepts, allowing you to add interactions and animations to your pages—even if previous attempts at writing JavaScript have left you baffled. This book guides you past the pitfalls associated with AJAX, events, effects, and advanced JavaScript language features.

A working demo of the examples in this book is available at:

<http://book.learningjquery.com>

For More Information: www.PacktPub.com/jquery/book

What This Book Covers

The first part of the book introduces jQuery and helps you to understand what the fuss is all about. *Chapter 1* covers downloading and setting up the jQuery library, as well as writing your first script.

The second part of the book steps you through each of the major aspects of the jQuery library. In *Chapter 2*, you'll learn how to get anything you want. The selector expressions in jQuery allow you to find elements on the page, wherever they may be. You'll work with these selector expressions to apply styling to a diverse set of page elements, sometimes in a way that pure CSS cannot.

In *Chapter 3*, you'll learn how to pull the trigger. You will use jQuery's event-handling mechanism to fire off behaviors when browser events occur. You'll also get the inside scoop on jQuery's secret sauce: attaching events unobtrusively, even before the page finishes loading.

In *Chapter 4*, you'll learn how to add flair to your actions. You'll be introduced to jQuery's animation techniques and see how to hide, show, and move page elements with the greatest of ease.

In *Chapter 5*, you'll learn how to change your page on command. This chapter will teach you how to alter the very structure an HTML document on the fly.

In *Chapter 6*, you'll learn how to make your site buzzword compliant. After reading this chapter, you, too, will be able to access server-side functionality without resorting to clunky page refreshes.

The third part of the book takes a different approach. Here you'll work through several real-world examples, pulling together what you've learned in previous chapters and creating robust jQuery solutions to common problems. In *Chapter 7*, you'll sort, sift, and style information to create beautiful and functional data layouts.

In *Chapter 8*, you'll master the finer points of client-side validation, design an adaptive form layout, and implement interactive client-server form features such as auto completion.

For More Information: www.PacktPub.com/jQuery/book
--

In *Chapter 9*, you'll enhance the beauty and utility of page elements by showing them in bite-size morsels. You'll make information fl y in and out of view both on its own and under user control.

In *Chapter 10* you'll learn about jQuery's impressive extension capabilities. You'll examine three prominent jQuery plug-ins and how to use them, and proceed to develop your own from the ground up.

Appendix A provides a handful of informative websites on a wide range of topics related to jQuery, JavaScript, and web development in general.

Appendix B recommends a number of useful third-party programs and utilities for editing and debugging jQuery code within your personal development environment.

Appendix C discusses one of the common stumbling blocks with the JavaScript language. You'll come to rely on the power of closures, rather than fear their side effects.

For More Information: www.PacktPub.com/jQuery/book

7

Table Manipulation

*Let 'em wear gaudy colors
Or avoid display*
– Devo,
"Wiggly World"

In the first six chapters, we explored the jQuery library in a series of tutorials that focused on each jQuery component and used examples as a way to see those components in action. In Chapters 7 through 9 we invert the process; we'll begin with the examples and see how we can use jQuery methods to achieve them.

Here we will use an online bookstore as our model website, but the techniques we cook up can be applied to a wide variety of other sites as well, from weblogs to portfolios, from market-facing business sites to corporate intranets. Chapters 7 and 8 focus on two common elements of most sites – tables and forms – while Chapter 9 examines a couple of ways to visually enhance sets of information using animated shufflers and rotators.

In this chapter, we will use jQuery to apply techniques for increasing the readability, usability, and visual appeal of tables, though we are not dealing with tables used for layout and design. In fact, as the web standards movement has become more pervasive in the last few years, table-based layout has increasingly been abandoned in favor of CSS-based designs. Although tables were often employed as a somewhat necessary stopgap measure in the 1990s to create multi-column and other complex layouts, they were never intended to be used in that way, whereas CSS is a technology expressly created for presentation.

But this is not the place for an extended discussion on the proper role of tables. Suffice it to say that in this chapter we will explore ways to display and interact with tables used as semantically marked up containers of tabular data. For a closer look at applying semantic, accessible HTML to tables, a good place to start is Roger Johansson's blog entry, *Bring on the Tables* at http://www.456bereastreet.com/archive/200410/bring_on_the_tables/.

For More Information: www.PacktPub.com/jquery/book

Some of the techniques we apply to tables in this chapter can be found in plug-ins such as Christian Bach's *Table Sorter*. For more information, visit the *jQuery Plug-in Repository* at <http://jquery.com/Plugins>.

Sorting

One of the most common tasks performed with tabular data is **sorting**. In a large table, being able to rearrange the information that we're looking for is invaluable. Unfortunately, this helpful operation is one of the trickiest to put into action. We can achieve the goal of sorting in two ways, namely Server-Side Sorting and JavaScript Sorting.

Server-Side Sorting

A common solution for data sorting is to perform it on the server side. Data in tables often comes from a database, which means that the code that pulls it out of the database can request it in a given sort order (using, for example, the SQL language's `ORDER BY` clause). If we have server-side code at our disposal, it is straightforward to begin with a reasonable default sort order.

Sorting is most useful when the user can determine the sort order. A common idiom is to make the headers of sortable columns into links. These links can go to the current page, but with a query string appended indicating the column to sort by:

```
<table id="my-data">
  <tr>
    <th class="name"><a href="index.php?sort=name">Name</a></th>
    <th class="date"><a href="index.php?sort=date">Date</a></th>
  </tr>
  ...
</table>
```

The server can react to the query string parameter by returning the database contents in a different order.

Preventing Page Refreshes

This setup is simple, but requires a page refresh for each sort operation. As we have seen, jQuery allows us to eliminate such page refreshes by using AJAX methods. If we have the column headers set up as links as before, we can add jQuery code to change those links into AJAX requests:

```
$(document).ready(function() {
  $('#my-data .name a').click(function() {
```

```

        $('#my-data').load('index.php?sort=name&type=ajax');
        return false;
    });
    $('#my-data .date a').click(function() {
        $('#my-data').load('index.php?sort=date&type=ajax');
        return false;
    });
});

```

Now when the anchors are clicked, jQuery sends an AJAX request to the server for the same page. We add an additional parameter to the query string so that the server can determine that an AJAX request is being made. The server code can be written to send back only the table itself, and not the surrounding page, when this parameter is present. This way we can take the response and insert it in place of the table.

This is an example of **progressive enhancement**. The page works perfectly well without any JavaScript at all, as the links for server-side sorting are still present. When JavaScript is present, however, the AJAX hijacks the page request and allows the sort to occur without a full page load.

JavaScript Sorting

There are times, though, when we either don't want to wait for server responses when sorting, or don't have a server-side scripting language available to us. A viable alternative in this case is to perform the sorting entirely on the browser using JavaScript client-side scripting.

For example, suppose we have a table listing books, along with their authors, release dates, and prices:

```

<table class="sortable">
  <thead>
    <tr>
      <th></th>
      <th>Title</th>
      <th>Author(s)</th>
      <th>Publish&nbsp;Date</th>
      <th>Price</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>
        <img src="../../covers/small/1847192386.png" width="49"
          height="61" alt="Building Websites with

```



```

Joomla! 1.5 Beta 1" />
    </td>
    <td>Building Websites with Joomla! 1.5 Beta 1</td>
    <td>Hagen Graf</td>
    <td>Feb 2007</td>
    <td>$40.49</td>
</tr>
<tr>
    <td></td>
    <td>Learning Mambo: A Step-by-Step Tutorial to Building Your
        Website</td>
    <td>Douglas Paterson</td>
    <td>Dec 2006</td>
    <td>$40.49</td>
</tr>
...
</tbody>
</table>
```





We'd like to turn the table headers into buttons that sort by their respective columns. Let us look into ways of doing this.

Row Grouping Tags

Note our use of the `<thead>` and `<tbody>` tags to segment the data into row groupings. Many HTML authors omit these implied tags, but they can prove useful in supplying us with more convenient CSS selectors to use. For example, suppose we wish to apply typical even/odd row striping to this table, but only to the body of the table:

```
$(document).ready(function() {
    $('table.sortable tbody tr:odd').addClass('odd');
    $('table.sortable tbody tr:even').addClass('even');
});
```

This will add alternating colors to the table, but leave the header untouched:

	Title	Author(s)	Publish Date	Price
	Building Websites with Joomla! 1.5 Beta 1	Hagen Graf	Feb 2007	\$40.49
	Learning Mambo: A Step-by-Step Tutorial to Building Your Website	Douglas Paterson	Dec 2006	\$40.49
	Moodle E-Learning Course Development	William Rice	May 2006	\$35.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Cherecheș-Toșa, Bogdan Brinzarea	Mar 2006	\$31.49

Basic Alphabetical Sorting

Now let's perform a sort on the **Title** column of the table. We'll need a class on the table header cell so that we can select it properly:

```
<thead>
  <tr>
    <th></th>
    <th class="sort-alpha">Title</th>
    <th>Author(s)</th>
    <th>Publish Date</th>
    <th>Price</th>
  </tr>
</thead>
```

To perform the actual sort, we can use JavaScript's built in `.sort()` method. It does an in-place sort on an array, and can take a function as an argument. This function compares two items in the array and should return a positive or negative number depending on the result. Our initial sort routine looks like this:

```
$(document).ready(function() {
  $('table.sortable').each(function() {
    var $table = $(this);
    $('th', $table).each(function(column) {
      if ($(this).is('.sort-alpha')) {
```

```
$(this).addClass('clickable').hover(function() {
    $(this).addClass('hover');
}, function() {
    $(this).removeClass('hover');
}).click(function() {
    var rows = $table.find('tbody > tr').get();
    rows.sort(function(a, b) {
        var keyA = $(a).children('td').eq(column).text()
                                                         .toUpperCase();
        var keyB = $(b).children('td').eq(column).text()
                                                         .toUpperCase();

        if (keyA < keyB) return -1;
        if (keyA > keyB) return 1;
        return 0;
    });
    $.each(rows, function(index, row) {
        $table.children('tbody').append(row);
    });
});
});
});
});
```





The first thing to note is our use of the `.each()` method to make iteration explicit. Even though we could bind a click handler to all headers with the `sort-alpha` class just by calling `$('#table.sortable th.sort-alpha').click()`, this wouldn't allow us to easily capture a crucial bit of information—the column index of the clicked header. Because `.each()` passes the iteration index into its callback function, we can use it to find the relevant cell in each row of the data later.

Once we have found the header cell, we retrieve an array of all of the data rows. This is a great example of how `.get()` is useful in transforming a jQuery object into an array of DOM nodes; even though jQuery objects act like arrays in many respects, they don't have any of the native array methods available, such as `.sort()`.

With `.sort()` at our disposal, the rest is fairly straightforward. The rows are sorted by comparing the textual contexts of the relevant table cell. We know which cell to look at because we captured the column index in the enclosing `.each()` call. We convert the text to uppercase because string comparisons in JavaScript are case-sensitive and we wish our sort to be case-insensitive. Finally, with the array sorted, we loop through the rows and reinsert them into the table. Since `.append()` does not clone nodes, this moves them rather than copying them. Our table is now sorted.

This is an example of progressive enhancement's counterpart, **graceful degradation**. Unlike with the AJAX solution discussed earlier, we cannot make the sort work without JavaScript, as we are assuming the server has no scripting language available to it in this case. The JavaScript is required for the sort to work, so by adding the "clickable" class only through code, we make sure not to indicate with the interface that sorting is even possible unless the script can run. The page *degrades* into one that is still functional, albeit without sorting available.

We have moved the actual rows around, hence our alternating row colors are now out of whack:

	Title	Author(s)	Publish Date	Price
	Advanced Microsoft Content Management Server Development	Angus Logan, Stefan Goßner, Lim Mei Ying, Andrew Connell	Nov 2005	\$53.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Cherecheș-Toșa, Bogdan Brinzarea	Mar 2006	\$31.49
	Alfresco Enterprise Content Management Implementation	Munwar Shariff	Jan 2007	\$53.99
	BPEL Cookbook: Best Practices for SOA-based integration and composite applications development	Jerry Thomas, Doug Todd, Harish Gaur, Lawrence Pravin, Arun Poduval, The Hoa Nguyen, Yves Coene, Jeremy Bolle, Stany Blanvalet, Markus Zirn, Matjaz Juric, Sean Carey, Michael Cardella, Kevin Geminiuc, Praveen Ramachandran	Jul 2006	\$40.49

We need to reapply the row colors after the sort is performed. We can do this by pulling the coloring code out into a function that we call when needed:

```
$(document).ready(function() {
    var alternateRowColors = function($table) {
        $('tbody tr:odd', $table).removeClass('even').addClass('odd');
        $('tbody tr:even', $table).removeClass('odd').addClass('even');
    };

    $('table.sortable').each(function() {
        var $table = $(this);
        alternateRowColors($table);
        $('th', $table).each(function(column) {
            if ($(this).is('.sort-alpha')) {
                $(this).addClass('clickable').hover(function() {
                    $(this).addClass('hover');
                }, function() {

```





```

$(this).removeClass('hover');
}).click(function() {
    var rows = $table.find('tbody > tr').get();
    rows.sort(function(a, b) {
        var keyA = $(a).children('td').eq(column).text()
            .toUpperCase();
        var keyB = $(b).children('td').eq(column).text()
            .toUpperCase();

        if (keyA < keyB) return -1;
        if (keyA > keyB) return 1;
        return 0;
    });
    $.each(rows, function(index, row) {
        $table.children('tbody').append(row);
    });
    alternateRowColors($table);
});
}
});
});
});

```

This corrects the row coloring after the fact, fixing our issue:

	Title	Author(s)	Publish Date	Price
	Advanced Microsoft Content Management Server Development	Angus Logan, Stefan Goßner, Lim Mei Ying, Andrew Connell	Nov 2005	\$53.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Cherecheș-Toșa, Bogdan Brinzarea	Mar 2006	\$31.49
	Alfresco Enterprise Content Management Implementation	Munwar Shariff	Jan 2007	\$53.99
	BPEL Cookbook: Best Practices for SOA-based Integration and composite applications development	Jerry Thomas, Doug Todd, Harish Gaur, Lawrence Pravin, Arun Poduval, The Hoa Nguyen, Yves Coene, Jeremy Bolie, Stany Blanvalet, Markus Zirn, Matjaz Juric, Sean Carey, Michael Cardella, Kevin Geminiuc, Praveen Ramachandran	Jul 2006	\$40.49

The Power of Plug-ins

The `alternateRowColors()` function that we wrote is a perfect candidate to become a jQuery plug-in. In fact, any operation that we wish to apply to a set of DOM elements can easily be expressed as a plug-in. In this case, we only need to modify our existing function a little bit:

```
jQuery.fn.alternateRowColors = function() {  
    $('tbody tr:odd', this).removeClass('even').addClass('odd');  
    $('tbody tr:even', this).removeClass('odd').addClass('even');  
    return this;  
};
```

We have made three important changes to the function.

- It is defined as a new property of `jQuery.fn` rather than as a standalone function. This registers the function as a plug-in method.
- We use the keyword `this` as a replacement for our `$table` parameter. Within a plug-in method, `this` refers to the jQuery object that is being acted upon.
- Finally, we return `this` at the end of the function. The return value makes our new method chainable.

More information on writing jQuery plug-ins can be found in Chapter 10. There we will discuss making a plug-in ready for public consumption, as opposed to the small example here that is only to be used by our own code.

With our new plug-in defined, we can call `$table.alternateRowColors()`, which is a more natural jQuery syntax, instead of `alternateRowColors($table)`.

Performance Concerns

Our code works, but is quite slow. The culprit is the comparator function, which is performing a fair amount of work. This comparator will be called many times during the course of a sort, which means that every extra moment it spends on processing will be magnified.

The actual sort algorithm used by JavaScript is not defined by the standard. It may be a simple sort like a bubble sort (worst case of $\Theta(n^2)$ in computational complexity terms) or a more sophisticated approach like quick sort (which is $\Theta(n \log n)$ on average). In either case doubling the number of items increases the number of times the comparator function is called by more than double.

The remedy for our slow comparator is to pre-compute the keys for the comparison. We begin with the slow sort function:

```
rows.sort(function(a, b) {
  keyA = $(a).children('td').eq(column).text().toUpperCase();
  keyB = $(b).children('td').eq(column).text().toUpperCase();
  if (keyA < keyB) return -1;
  if (keyA > keyB) return 1;
  return 0;
});
$.each(rows, function(index, row) {
  $table.children('tbody').append(row);
});
```

We can pull out the key computation and do that in a separate loop:

```
$.each(rows, function(index, row) {
  row.sortKey = $(row).children('td').eq(column).text().toUpperCase();
});
rows.sort(function(a, b) {
  if (a.sortKey < b.sortKey) return -1;
  if (a.sortKey > b.sortKey) return 1;
  return 0;
});
$.each(rows, function(index, row) {
  $table.children('tbody').append(row);
  row.sortKey = null;
});
```

In the new loop, we are doing all of the expensive work and storing the result in a new property. This kind of property, attached to a DOM element but not a normal DOM attribute, is called an **expando**. This is a convenient place to store the key since we need one per table row element. Now we can examine this attribute within the comparator function, and our sort is markedly faster.



We set the expando property to null after we're done with it to clean up after ourselves. This is not necessary in this case, but is a good habit to establish because expando properties left lying around can be the cause of memory leaks. For more information, see Appendix C.

Finessing the Sort Keys





Now we want to apply the same kind of sorting behavior to the **Author(s)** column of our table. By adding the `sort-alpha` class to its table header cell, the **Author(s)** column can be sorted with our existing code. But ideally authors should be sorted by last name, not first. Since some books have multiple authors, and some authors have middle names or initials listed, we need outside guidance to determine what part of the text to use as our sort key. We can supply this guidance by wrapping the relevant part of the cell in a tag:

```
<tr>
  <td>
    </td>
    <td>Building Websites with Joomla! 1.5 Beta 1</td>
    <td>Hagen <span class="sort-key">Graf</span></td>
    <td>Feb 2007</td>
    <td>$40.49</td>
  </tr>
  <tr>
    <td>
      </td>
    <td>Learning Mambo: A Step-by-Step Tutorial to Building Your Website
    </td>
    <td>Douglas <span class="sort-key">Paterson</span></td>
    <td>Dec 2006</td>
    <td>$40.49</td>
  </tr>
  <tr>
    <td>
      </td>
    <td>Moodle E-Learning Course Development</td>
    <td>William <span class="sort-key">Rice</span></td>
    <td>May 2006</td>
    <td>$35.99</td>
  </tr>
```

Now we have to modify our sorting code to take this tag into account, without disturbing the existing behavior for the **Title** column, which is working well. By prepending the marked sort key to the key we have previously calculated, we can sort first on the last name if it is called out, but on the whole string as a fallback:


```
$.each(rows, function(index, row) {
    var $cell = $(row).children('td').eq(column);
    row.sortKey = $cell.find('.sort-key').text().toUpperCase()
        + ' ' + $cell.text().toUpperCase();
});
```

Sorting by the **Author(s)** column now uses the last name:

	Title	Author(s)	Publish Date	Price
	Programming Windows Workflow Foundation: Practical WF Techniques and Examples using XAML and C#	K. Scott Allen	Dec 2006	\$40.49
	Building Websites with XOOPS: A step-by-step tutorial	Steve Atwal	Oct 2006	\$26.99
	Learn OpenOffice.org Spreadsheet Macro Programming: OOoBasic and Calc automation	Dr. Mark Alexander Bain	Dec 2006	\$35.99
	UML 2.0 in Action: A project-based tutorial	Philippe Baumann, Henriette Baumann, Patrick Grassle	Sep 2005	\$31.49

If two last names are identical, the sort uses the entire string as a tiebreaker for positioning.

Sorting Other Types of Data

Our sort routine should be able to handle not just the **Title** and **Author** columns, but the **Publish Dates** and **Price** as well. Since we streamlined our comparator function, it can handle all kinds of data, but the computed keys will need to be adjusted for other data types. For example, in the case of prices we need to strip off the leading \$ character and parse the rest, then compare them:

```
var key = parseFloat($cell.text().replace(/^[^\d.]*/, ''));
row.sortKey = isNaN(key) ? 0 : key;
```

The result of `parseFloat()` needs to be checked, because if no number can be extracted from the text, `NaN` is returned, which can wreak havoc on `.sort()`. For the date cells, we can use the JavaScript `Date` object:

```
row.sortKey = Date.parse('1 ' + $cell.text());
```

The dates in this table contain a month and year only; `Date.parse()` requires a fully-specified date, so we prepend the string with 1. This provides a day to complement the month and year, and the combination is then converted into a timestamp, which can be sorted using our normal comparator.

We can apportion these expressions across separate functions, and call the appropriate one based on the class applied to the table header:

```
$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};

$(document).ready(function() {
    var alternateRowColors = function($table) {
        $('tbody tr:odd', $table).removeClass('even').addClass('odd');
        $('tbody tr:even', $table).removeClass('odd').addClass('even');
    };





    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors($table);
        $('th', $table).each(function(column) {
            var findSortKey;
            if ($(this).is('.sort-alpha')) {
                findSortKey = function($cell) {
                    return $cell.find('.sort-key').text().toUpperCase()
                        + ' ' + $cell.text().toUpperCase();
                };
            }
            else if ($(this).is('.sort-numeric')) {
                findSortKey = function($cell) {
                    var key = parseFloat($cell.text().replace(/^[^\.d.]*/, ''));
                    return isNaN(key) ? 0 : key;
                };
            }
            else if ($(this).is('.sort-date')) {
                findSortKey = function($cell) {
                    return Date.parse('1 ' + $cell.text());
                };
            }
            if (findSortKey) {
                $(this).addClass('clickable').hover(function() {
                    $(this).addClass('hover');
                }, function() {
                    $(this).removeClass('hover');
                });
            }
        });
    });
});
```

```

    }).click(function() {
        var rows = $table.find('tbody > tr').get();
        $.each(rows, function(index, row) {
            row.sortKey =
                findSortKey($(row).children('td').eq(column));
        });
        rows.sort(function(a, b) {
            if (a.sortKey < b.sortKey) return -1;
            if (a.sortKey > b.sortKey) return 1;
            return 0;
        });
        $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
        });
        $table.alternateRowColors($table);
    });
}
});
});
});

```

The `findSortKey` variable doubles as the function to calculate the key and a flag to indicate whether the column header is marked with a class making it sortable. We can now sort on date or price:





	Title	Author(s)	Publish Date	Price
	User Training for Busy Programmers	William Rice	Jun 2005	\$11.69
	Pluggable Authentication Modules: The Definitive Guide to PAM for Linux SysAdmins and C Developers	Kenneth Geissshirt	Jan 2007	\$17.99
	Creating your MySQL Database: Practical Design Tips and Techniques	Marc Delisle	Nov 2006	\$17.99
	The Microsoft Outlook Ideas Book	Barbara March	Mar 2006	\$22.49

Column Highlighting

It can be a nice user interface enhancement to visually remind the user of what has been done in the past. By highlighting the column that was most recently used for sorting, we can focus the user's attention on the part of the table that is most likely to be relevant. Fortunately, since we've already determined how to select the table cells in the column, applying a class to those cells is simple:

```
$table.find('td').removeClass('sorted')
    .filter(':nth-child(' + (column + 1) + ')').addClass('sorted');
```

Note that we have to add one to the column index we found earlier, since the `:nth-child()` selector is *one-based* rather than *zero-based*. With this code in place, we get a highlighted column after any sort operation:

	Title	Author(s)	Publish Date	Price
	Building Websites with the ASP.NET Community Starter Kit	Cristian Darie, K. Scott Allen	May 2004	\$40.49
	Building Websites with Plone	Cameron Cooper	Nov 2004	\$44.99
	Windows Server 2003 Active Directory Design and Implementation: Creating, Migrating, and Merging Networks	John Savill	Jan 2005	\$53.99
	SSL VPN: Understanding, evaluating and planning secure, web-based remote access	Tim Speed, Joseph Steinberg	Mar 2005	\$44.99

Alternating Sort Directions

Our final sorting enhancement is to allow for both ascending and descending sort orders. When the user clicks on a column that is already sorted, we want to reverse the current sort order.

To reverse a sort, all we have to do is to invert the values returned by our comparator. We can do this with a simple variable:

```
if (a.sortKey < b.sortKey) return -newDirection;
if (a.sortKey > b.sortKey) return newDirection;
```

If `newDirection` equals 1, then the sort will be the same as before. If it equals -1, the sort will be reversed. We can use classes to keep track of the current sort order of a column:

```
$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};

$(document).ready(function() {
    var alternateRowColors = function($table) {
        $('tbody tr:odd', $table).removeClass('even').addClass('odd');
        $('tbody tr:even', $table).removeClass('odd').addClass('even');
    };
    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors($table);
        $('th', $table).each(function(column) {
            var findSortKey;
            if ($(this).is('.sort-alpha')) {
                findSortKey = function($cell) {
                    return $cell.find('.sort-key').text().toUpperCase() + ' ' +
                        $cell.text().toUpperCase();
                };
            }
            else if ($(this).is('.sort-numeric')) {
                findSortKey = function($cell) {
                    var key = parseFloat($cell.text().replace(/^[^\d.]*/, ''));
                    return isNaN(key) ? 0 : key;
                };
            }
            else if ($(this).is('.sort-date')) {
                findSortKey = function($cell) {
                    return Date.parse('1 ' + $cell.text());
                };
            }
            if (findSortKey) {
                $(this).addClass('clickable').hover(function() {
                    $(this).addClass('hover');
                }, function() {
                    $(this).removeClass('hover');
                }).click(function() {
                    var newDirection = 1;

```





```

        if ($(this).is('.sorted-asc')) {
            newDirection = -1;
        }
        var rows = $table.find('tbody > tr').get();

        $.each(rows, function(index, row) {
            row.sortKey =
                findSortKey($(row).children('td').eq(column));
        });
        rows.sort(function(a, b) {
            if (a.sortKey < b.sortKey) return -newDirection;
            if (a.sortKey > b.sortKey) return newDirection;
            return 0;
        });
        $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
        });
        $table.find('th').removeClass('sorted-asc')
            .removeClass('sorted-desc');
        var $sortHead = $table.find('th').filter('
            :nth-child(' + (column + 1) + ')');
        if (newDirection == 1) {
            $sortHead.addClass('sorted-asc');
        } else {
            $sortHead.addClass('sorted-desc');
        }
        $table.find('td').removeClass('sorted')
            .filter(':nth-child(' + (column + 1) + ')')
                .addClass('sorted');
        $table.alternateRowColors($table);
    });
}
});
});
});

```

As a side benefit, since we use classes to store the sort direction we can style the columns headers to indicate the current order as well:

	Title	Author(s)	Publish Date	Price
	Enhancing Microsoft Content Management Server with ASP.NET 2.0	Lim Mei Ying, Spencer Harbar, Stefan Goßner	Aug 2006	\$34.19
	Openswan: Building and Integrating Virtual Private Networks	Paul Wouters, Ken Bantoft	Feb 2006	\$53.99
	Learning Jakarta Struts 1.2: a concise and practical tutorial	Stephan Wiesner	Aug 2005	\$31.49
	Implementing SugarCRM	Michael J.R. Whitehead	Feb 2006	\$44.99

Pagination

Sorting is a great way to wade through a large amount of data to find information. We can also help the user focus on a portion of a large data set by paginating the data. Pagination can be done in two ways—Server-Side Pagination and JavaScript Pagination.

Server-Side Pagination

Much like sorting, **pagination** is often performed on the server. If the data to be displayed is stored in a database, it is easy to pull out one chunk of information at a time using MySQL's `LIMIT` clause, `ROWNUM` in Oracle, or equivalent methods in other database engines.

As with our initial sorting example, pagination can be triggered by sending information to the server in a query string, such as `index.php?page=52`. And again as before, we can perform this task either with a full page load or by using AJAX to pull in just one chunk of the table. This strategy is browser-independent, and can handle large data sets very well.

Sorting and Paging Go Together

Data that is long enough to benefit from sorting is likely long enough to be a candidate for paging. It is not unusual to wish to combine these two techniques for data presentation. Since they both affect the set of data that is present on a page, though, it is important to consider their interactions while implementing them.

Both sorting and pagination can be accomplished either on the server or in the web browser. However, we must keep the strategies for the two tasks in sync; otherwise, we can end up with confusing behavior. Suppose, for example, that both sorting and paging is done on the server:

A	4
B	5
C	2
D	7
E	1
F	8
G	3
H	6

E	1
C	2
G	3
A	4
B	5
H	6
D	7
F	8

When the table is re-sorted by number, a different set of rows is present on **Page 1** of the table. If paging is done by the server and sorting by the browser, the entire data set is not available for the sorting routine, making the results incorrect:

A	4
B	5
C	2
D	7

C	2
A	4
B	5
D	7

Only the data already present on the page can be displayed. To prevent this from being a problem, we must either perform both tasks on the server, or both in the browser.

JavaScript Pagination

So, let's examine how we would add JavaScript pagination to the table we have already made sortable in the browser. First, we'll focus on displaying a particular page of data, disregarding user interaction for now:


```
$(document).ready(function() {
  $('table.paginated').each(function() {
    var currentPage = 0;
    var numPerPage = 10;
    var $table = $(this);
    $table.find('tbody tr').show()
      .lt(currentPage * numPerPage)
      .hide()
      .end()
      .gt((currentPage + 1) * numPerPage - 1)
      .hide()
      .end();
  });
});
```

This code displays the first page—ten rows of data.

Once again we rely on the presence of a `<tbody>` element to separate data from headers; we don't want to have the headers or footers disappear when moving on to the second page. For selecting the rows containing data, we show all the rows first, then select the rows before and after the current page, hiding them. The method chaining supported by jQuery makes another appearance here when we filter the set of matched rows twice, using `.end()` in between to *pop* the current filter off the stack and start afresh with a new filter.

The most error-prone task in writing this code is formulating the expressions to use in the filters. To use the `.lt()` and `.gt()` methods, we need to find the indices of the rows at the beginning and end of the current page. For the beginning row, we just multiply the current page number by the number of rows on each page. Multiplying the number of rows by one more than the current page number gives us the *beginning* row of the *next* page; to find the *last* row of the *current* page, we must subtract one from this.

Displaying the Pager

To add user interaction to the mix, we need to place the **pager** itself next to the table. We could do this by simply inserting links for the pages in the HTML markup, but this would violate the progressive enhancement principle we've been espousing. Instead, we should add the links using JavaScript, so that users without scripting available are not misled by links that cannot work.

To display the links, we need to calculate the number of pages and create a corresponding number of DOM elements:

```





var numRows = $table.find('tbody tr').length;
var numPages = Math.ceil(numRows / numPerPage);

var $pager = $('<div class="pager"></div>');
for (var page = 0; page < numPages; page++) {
    $('<span class="page-number">' + (page + 1) + '</span>')
        .appendTo($pager).addClass('clickable');
}
$pager.insertBefore($table);

```

The number of pages can be found by dividing the number of data rows by the number of items we wish to display on each page. If the division does not yield an integer, we must round the result up using `Math.ceil()` to ensure that the final partial page will be accessible. Then, with this number in hand, we create buttons for each page and position the new pager before the table:

1 2 3 4 5 6 7

	Title	Author(s)	Publish Date	Price
	Building Websites with Joomla! 1.5 Beta 1	Hagen Graf	Feb 2007	\$40.49
	Learning Mambo: A Step-by-Step Tutorial to Building Your Website	Douglas Paterson	Dec 2006	\$40.49
	Moodle E-Learning Course Development	William Rice	May 2006	\$35.99
	AJAX and PHP: Building Responsive Web Applications	Cristian Darie, Mihai Bucica, Filip Cherecheș-Toșa, Bogdan Brinzarea	Mar 2006	\$31.49

Enabling the Pager Buttons

To make these new buttons actually work, we need to update the `currentPage` variable and then run our pagination routine. At first blush, it seems we should be able to do this by setting `currentPage` to `page`, which is the current value of the iterator that creates the buttons:

```

$(document).ready(function() {
    $('table.paginated').each(function() {
        var currentPage = 0;
        var numPerPage = 10;
        var $table = $(this);

```

```
var repaginate = function() {
    $table.find('tbody tr').show()
        .lt(currentPage * numPerPage)
        .hide()
    .end()
    .gt((currentPage + 1) * numPerPage - 1)
        .hide()
    .end();
};
var numRows = $table.find('tbody tr').length;
var numPages = Math.ceil(numRows / numPerPage);
var $pager = $('<div class="pager"></div>');
for (var page = 0; page < numPages; page++) {
    $('<span class="page-number">' + (page + 1) + '</span>')
        .click(function() {
            currentPage = page;
            repaginate();
        })
        .appendTo($pager).addClass('clickable');
}
$pager.insertBefore($table);
repaginate();
});
```

This mostly works. The new `repaginate()` function is called when the page loads and when any button is clicked. All of the buttons take us to a page with no rows on it, though:

1	2	3	4	5	6	7
	Title	Author(s)	Publish Date	Price		

The problem is that in defining our click handler, we have created a **closure**. The click handler refers to the `page` variable, which is defined outside the function. When the variable changes the next time through the loop, this affects the click handlers that we have already set up for the earlier buttons. The net effect is that, for a pager with 7 pages, each button directs us to page 8 (the final value of `page`). More information on how closures work can be found in Appendix C, *JavaScript Closures*.

To correct this problem, we'll take advantage of one of the more advanced features of jQuery's event binding methods. We can add a set of data to the handler when we bind it that will still be available when the handler is eventually called. With this capability in our bag of tricks, we can write:


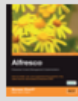


```

$('

```

The new page number is passed into the handler by way of the event's data property. In this way the page number escapes the closure, and is frozen in time at the value it contained when the handler was bound. Now our pager buttons can correctly take us to different pages:

1 2 3 4 5 6 7

	Title	Author(s)	Publish Date	Price
	PHPEclipse: A User Guide	Shu-Wai Chow	Feb 2006	\$31.49
	Alfresco Enterprise Content Management Implementation	Munwar Shariff	Jan 2007	\$53.99
	Smarty PHP Template Programming and Applications	Joao Prado Maia, Hasin Hayder, Lucian Gheorghe	Apr 2006	\$35.99
	JasperReports for Java Developers	David Heffelfinger	Aug 2006	\$40.49

Marking the Current Page

Our pager can be made more user-friendly by highlighting the current page number. We just need to update the classes on the buttons every time one is clicked:

```

var $pager = $('

[ 157 ]







For More Information: www.PacktPub.com/jQuery/book


```

Now we have an indicator of the current status of the pager:

1 2 3 4 5 6 7

	Title	Author(s)	Publish Date	Price
	Linux Email: Set up and Run a Small Office Email Server	Patrick Ben Koetter, Carl Taylor, Ralf Hildebrandt, David Rusenko, Alistair McDonald, Magnus Back	Jul 2005	\$35.99
	Windows Small Business Server SBS 2003: A Clear and Concise Administrator's Reference and How-To	Stephanie Knecht-Thurmann	Aug 2005	\$33.29
	Creating your MySQL Database: Practical Design Tips and Techniques	Marc Delisle	Nov 2006	\$17.99
	Building Websites with VB.NET and DotNetNuke 4	Michael Washington, Steve Valenzuela, Daniel N. Egan	Oct 2006	\$35.99

Paging with Sorting

We began this discussion by noting that sorting and paging controls needed to be aware of one another to avoid confusing results. Now that we have a working pager, we need to make sort operations respect the current page selection.

Doing this is as simple as calling our `repaginate()` function whenever a sort is performed. The scope of the function, though, makes this problematic. We can't reach `repaginate()` from our sorting routine because it is contained inside a different `$(document).ready()` handler. We could just consolidate the two pieces of code, but instead let's be a bit sneakier. We can decouple the behaviors, so that a sort calls the `repaginate` behavior if it exists, but ignores it otherwise. To accomplish this, we'll use a handler for a custom event.

In our earlier event handling discussion, we limited ourselves to event names that were triggered by the web browser, such as `click` and `mouseup`. The `.bind()` and `.trigger()` methods are not limited to these events, though; we can use any string as an event name. In this case, we can define a new event called `repaginate` as a stand-in for the function we've been calling:

```
$table.bind('repaginate', function() {
    $table.find('tbody tr').show()
    .lt(currentPage * numPerPage)
    .hide()
    .end()
})
```

```

        .gt((currentPage + 1) * numPerPage - 1)
        .hide()
        .end();
    });

```

Now in places where we were calling `repaginate()`, we can call:

```
$table.trigger('repaginate');
```

We can issue this call in our sort code as well. It will do nothing if the table does not have a pager, so we can mix and match the two capabilities as desired.

The Finished Code

The completed sorting and paging code in its entirety follows:

```

$.fn.alternateRowColors = function() {
    $('tbody tr:odd', this).removeClass('even').addClass('odd');
    $('tbody tr:even', this).removeClass('odd').addClass('even');
    return this;
};

$(document).ready(function() {
    var alternateRowColors = function($table) {
        $('tbody tr:odd', $table).removeClass('even').addClass('odd');
        $('tbody tr:even', $table).removeClass('odd').addClass('even');
    };

    $('table.sortable').each(function() {
        var $table = $(this);
        $table.alternateRowColors($table);
        $table.find('th').each(function(column) {
            var findSortKey;

            if ($(this).is('.sort-alpha')) {
                findSortKey = function($cell) {
                    return $cell.find('.sort-key').text().toUpperCase() +
                        ' ' + $cell.text().toUpperCase();
                };
            }
            else if ($(this).is('.sort-numeric')) {
                findSortKey = function($cell) {
                    var key = parseFloat($cell.text().replace(/^[^\d.]*\/, ''));
                    return isNaN(key) ? 0 : key;
                };
            }
        });
    });

```

```
    }
    else if ($(this).is('.sort-date')) {
        findSortKey = function($cell) {
            return Date.parse('1 ' + $cell.text());
        };
    }
}

if (findSortKey) {
    $(this).addClass('clickable').hover(function() {
        $(this).addClass('hover');
    }, function() {
        $(this).removeClass('hover');
    }).click(function() {
        var newDirection = 1;
        if ($(this).is('.sorted-asc')) {
            newDirection = -1;
        }

        rows = $table.find('tbody > tr').get();

        $.each(rows, function(index, row) {
            row.sortKey =
                findSortKey($(row).children('td').eq(column));
        });

        rows.sort(function(a, b) {
            if (a.sortKey < b.sortKey) return -newDirection;
            if (a.sortKey > b.sortKey) return newDirection;
            return 0;
        });

        $.each(rows, function(index, row) {
            $table.children('tbody').append(row);
            row.sortKey = null;
        });

        $table.find('th').removeClass('sorted-asc')
            .removeClass('sorted-desc');
        var $sortHead = $table.find('th').filter(':nth-child('
            + (column + 1) + ')');

        if (newDirection == 1) {
            $sortHead.addClass('sorted-asc');
        } else {
            $sortHead.addClass('sorted-desc');
        }

        $table.find('td').removeClass('sorted')
            .filter(':nth-child(' + (column + 1) + ')')
            .addClass('sorted');
```

```

        $table.alternateRowColors($table);
        $table.trigger('repaginate');
    });
    }
    });
    });
    });
$(document).ready(function() {
    $('table.paginated').each(function() {
        var currentPage = 0;
        var numPerPage = 10;

        var $table = $(this);

        $table.bind('repaginate', function() {
            $table.find('tbody tr').show()
                .lt(currentPage * numPerPage)
                .hide()
            .end()
            .gt((currentPage + 1) * numPerPage - 1)
                .hide()
            .end();
        });

        var numRows = $table.find('tbody tr').length;
        var numPages = Math.ceil(numRows / numPerPage);

        var $pager = $('<div class="pager"></div>');
        for (var page = 0; page < numPages; page++) {
            $('<span class="page-number">' + (page + 1) + '</span>')
                .bind('click', {'newPage': page}, function(event) {
                    currentPage = event.data['newPage'];
                    $table.trigger('repaginate');
                    $(this).addClass('active').siblings().removeClass('active');
                })
            .appendTo($pager).addClass('clickable');
        }
        $pager.find('span.page-number:first').addClass('active');
        $pager.insertBefore($table);

        $table.trigger('repaginate');
    });
});

```


The rest of this Chapter covers:

Advanced Row Striping

Three-color Alternating Pattern

Alternating Triplets

Row Highlighting

Tooltips

Collapsing and Expanding

Filtering

Filter Options

Collecting Filter Options from Content

Reversing the Filters

Interacting with Other Code

Row Striping

Expanding and Collapsing

The Finished Code

Summary

In this chapter, we have explored some of the ways to slice and dice the tables on our sites, reconfiguring them into beautiful and functional containers for our data. We have covered sorting data in tables, using different kinds of data (words, numbers, dates) as sort keys along with paginating tables into easily-viewed chunks. We have learned sophisticated row striping techniques and JavaScript-powered tooltips. We have also walked through expanding and collapsing as well as filtering and highlighting of rows that match the given criteria.

We've even touched briefly on some quite advanced topics, such as sorting and paging with server-side code and AJAX techniques, dynamically calculating page coordinates for elements, and writing a jQuery plug-in.

As we have seen, properly semantic HTML tables wrap a great deal of subtlety and complexity in a small package. Fortunately, jQuery can help us easily tame these creatures, allowing the full power of tabular data to come to the surface.

Where to buy this book

You can buy Learning jQuery: Better Interaction Design and Web Development with Simple Javascript Techniques from the Packt Publishing website:

<http://www.packtpub.com/jquery/book>.

Free shipping to the US, UK, Europe, Australia, New Zealand and India.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



www.PacktPub.com

For More Information: www.PacktPub.com/jquery/book