1

Introducing jQuery

Today's World Wide Web is a dynamic environment, and its users set a high bar for both style and function of sites. To build interesting, interactive sites, developers are turning to JavaScript libraries such as jQuery to automate common tasks and simplify complicated ones. One reason the jQuery library is a popular choice is its ability to assist in a wide range of tasks.

Because jQuery does perform so many different functions, it can seem challenging to know where to begin. Yet, there is a coherence and symmetry to the design of the library; most of its concepts are borrowed from the structure of HTML and **Cascading Style Sheets** (**CSS**). Because many web developers have more experience with these technologies than with JavaScript, the library's design lends itself to a quick start for designers with little programming experience. In fact, in this opening chapter we'll write a functioning jQuery program in just three lines of code. On the other hand, experienced programmers will also be aided by this conceptual consistency, as we'll see in the later, more advanced chapters.

But before we illustrate the operation of the library with an example, we should discuss why we might need it in the first place.

What jQuery Does

The jQuery library provides a general-purpose abstraction layer for common web scripting, and is therefore useful in almost every scripting situation. Its extensible nature means that we could never cover all possible uses and functions in a single book, as plug-ins are constantly being developed to add new abilities. The core features, though, address the following needs:

Access parts of a page. Without a JavaScript library, many lines of code must be written to traverse the **Document Object Model** (**DOM**) tree, and locate specific portions of an HTML document's structure. jQuery offers a robust and efficient selector mechanism for retrieving exactly the piece of the document that is to be inspected or manipulated.

Modify the appearance of a page. CSS offers a powerful method of influencing the way a document is rendered; but it falls short when web browsers do not all support the same standards. jQuery can

bridge this gap, providing the same standards support across all browsers. In addition, jQuery can change the classes or individual style properties applied to a portion of the document even after the page has been rendered.

Alter the content of a page. Not limited to mere cosmetic changes, jQuery can modify the content of a document itself with a few keystrokes. Text can be changed, images can be inserted or swapped, lists can be reordered, or the entire structure of the HTML can be rewritten and extended—all with a single easy-to-use API.

Respond to a user's interaction with a page. Even the most elaborate and powerful behaviors are not useful if we can't control when they take place. The jQuery library offers an elegant way to intercept a wide variety of events, such as a user clicking on a link, without the need to clutter the HTML code itself with event handlers. At the same time, its event-handling API removes browser inconsistencies that often plague web developers.

Add animation to a page. To effectively implement such interactive behaviors, a designer must also provide visual feedback to the user. The jQuery library facilitates this by providing an array of effects such as fades and wipes, as well as a toolkit for crafting new ones.

Retrieve information from a server without refreshing a page. This code pattern has become known as **Asynchronous JavaScript and XML** (**AJAX**), and assists web developers in crafting a responsive, feature-rich site. The jQuery library removes the browser-specific complexity from this process, allowing developers to focus on the server-end functionality.

Simplify common JavaScript tasks. In addition to all of the document-specific features of jQuery, the library provides enhancements to basic JavaScript constructs such as iteration and array manipulation.

Why jQuery Works Well

With the recent resurgence of interest in dynamic HTML comes a proliferation of JavaScript frameworks. Some are specialized, focusing on just one or two of the above tasks. Others attempt to catalog every possible behavior and animation, and serve these all up pre-packaged. To maintain the wide range of features outlined above while remaining compact, jQuery employs several strategies:

Leverage knowledge of CSS. By basing the mechanism for locating page elements on CSS selectors, jQuery inherits a terse yet legible way of expressing a document's structure. Because a prerequisite for doing professional web development is knowledge of CSS syntax, jQuery becomes an entry point for designers who want to add behavior to their pages.

Support extensions. In order to avoid *feature creep*, jQuery relegates special-case uses to plug-ins. The method for creating new plug-ins is simple and well-documented, which has spurred the development of a wide variety of inventive and useful modules. Even most of the features in the basic jQuery download are internally realized through the plug-in architecture, and can be removed if desired, yielding an even smaller library.

Abstract away browser quirks. An unfortunate reality of web development is that each browser has its own set of deviations from published standards. A significant portion of any web application can be relegated to handling features differently on each platform. While the ever-evolving browser landscape makes a perfectly browser-neutral code base impossible for some advanced features, jQuery adds an abstraction layer that normalizes the common tasks, reducing the size of code, and tremendously simplifying it.

Always work with sets. When we instruct jQuery, *Find all elements with the class 'collapsible' and hide them*, there is no need to loop through each returned element. Instead, methods such as .hide() are designed to automatically work on sets of objects instead of individual ones. This technique, called **implicit iteration**, means that many looping constructs become unnecessary, shortening code considerably.

Allow multiple actions in one line. To avoid overuse of temporary variables or wasteful repetition, jQuery employs a programming pattern called **chaining** for the majority of its methods. This means that the result of most operations on an object is the object itself, ready for the next action to be applied to it.

These strategies have kept the jQuery package slim—roughly 20KB compressed—while at the same time providing techniques for keeping our custom code that uses the library compact, as well.

The elegance of the library comes about partly by design, and partly due to the evolutionary process spurred by the vibrant community that has sprung up around the project. Users of jQuery gather to discuss not only the development of plug-ins, but also enhancements to the core library. Appendix A details many of the community resources available to jQuery developers.

Despite all of the efforts required to engineer such a flexible and robust system, the end product is free for all to use. This open-source project is dually licensed under the **GNU Public License** (appropriate for inclusion in many other open-source projects) and the **MIT License** (to facilitate use of jQuery within proprietary software).

2

Selector

The jQuery library harnesses the power of Cascading Style Sheets (CSS) selectors to let us quickly and easily access elements or groups of elements in the Document Object Model (DOM).

A jQuery Selector is a function which makes use of expressions to find out matching elements from a DOM based on the given criteria.

The $() factory function:

All type of selectors available in jQuery, always start with the dollar sign and parentheses: **$()**.

The factory function **$()** makes use of following three building blocks while selecting elements in a given document:

jQuery	Description
Tag Name:	Represents a tag name available in the DOM. For example **$('p')** selects all paragraphs in the document.
Tag ID:	Represents a tag available with the given ID in the DOM. For example **$('#some-id')** selects the single element in the document that has an ID of some-id.
Tag Class:	Represents a tag available with the given class in the DOM. For example **$('.some-class')** selects all elements in the document that have a class of some-class.

All the above items can be used either on their own or in combination with other selectors. All the jQuery selectors are based on the same principle except some tweaking.

NOTE: The factory function **$()** is a synonym of **jQuery()** function. So in case you are using any other JavaScript library where **$** sign is conflicting with some thing else then you can replace **$** sign by **jQuery** name and you can use function **jQuery()** instead of **$()**.

Example: Following is a simple example which makes use of Tag Selector. This would select all the elements with a tag name **p**

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>

  <script type="text/javascript" language="javascript">
  $(document).ready(function() {
    var pars = $("p");
    for( i=0; i<pars.length; i++ ){
      alert("Found paragraph: " + pars[i].innerHTML);
    }
  });
  </script>
</head>
<body>
  <div>
    <p class="myclass">This is a paragraph.</p>
    <p id="myid">This is second paragraph.</p>
    <p>This is third paragraph.</p>
  </div>
</body>
</html>
```

How to use Selectors?

The selectors are very useful and would be required at every step while using jQuery. They get the exact element that you want from your HTML document.

Following table lists down few basic selectors and explains them with examples.

Selector	Description
Name	Selects all elements which match with the given element **Name**.
#ID	Selects a single element which matches with the given **ID**
.Class	Selects all elements which match with the given **Class**.
Universal (*)	Selects all elements available in a DOM.
Multiple Elements E, F, G	Selects the combined results of all the specified selectors **E, F** or **G**.

Similar to above syntax and examples, following examples would give you understanding on using different type of other useful selectors:

- **$('*'):** This selector selects all elements in the document.
- **$("p > *"):** This selector selects all elements that are children of a paragraph element.
- **$("#specialID"):** This selector function gets the element with id="specialID".
- **$(".specialClass"):** This selector gets all the elements that have the class of *specialClass*.
- **$("li:not(.myclass)"):** Selects all elements matched by <li> that do not have class="myclass".
- **$("a#specialID.specialClass"):** This selector matches links with an id of *specialID* and a class of *specialClass*.
- **$("p a.specialClass"):** This selector matches links with a class of *specialClass* declared within <p> elements.
- **$("ul li:first"):** This selector gets only the first <li> element of the <ul>.
- **$("#container p"):** Selects all elements matched by <p> that are descendants of an element that has an id of *container*.
- **$("li > ul"):** Selects all elements matched by <ul> that are children of an element matched by
- **$("strong + em"):** Selects all elements matched by that immediately follow a sibling element matched by .

- **$("p ~ ul"):** Selects all elements matched by <ul> that follow a sibling element matched by <p>.
- **$("code, em, strong"):** Selects all elements matched by <code> or or .
- **$("p strong, .myclass"):** Selects all elements matched by that are descendants of an element matched by <p> as well as all elements that have a class of *myclass*.
- **$(":empty"):** Selects all elements that have no children.
- **$("p:empty"):** Selects all elements matched by <p> that have no children.
- **$("div[p]"):** Selects all elements matched by that contain an element matched by <p>.
- **$("p[.myclass]"):** Selects all elements matched by <p> that contain an element with a class of *myclass*.
- **$("a[@rel]"):** Selects all elements matched by <a> that have a rel attribute.
- **$("input[@name=myname]"):** Selects all elements matched by <input> that have a name value exactly equal to *myname.*
- **$("input[@name^=myname]"):** Selects all elements matched by <input> that have a name value beginning with *myname*.
- **$("a[@rel$=self]"):** Selects all elements matched by <p> that have a class value ending with *bar*
- **$("a[@href*=domain.com]"):** Selects all elements matched by <a> that have an href value containing domain.com.
- **$("li:even"):** Selects all elements matched by <li> that have an even index value.
- **$("tr:odd"):** Selects all elements matched by <tr> that have an odd index value.
- **$("li:first"):** Selects the first <li> element.
- **$("li:last"):** Selects the last <li> element.
- **$("li:visible"):** Selects all elements matched by <li> that are visible.
- **$("li:hidden"):** Selects all elements matched by <li> that are hidden.
- **$(":radio"):** Selects all radio buttons in the form.
- **$(":checked"):** Selects all checked boxex in the form.
- **$(":input"):** Selects only form elements (input, select, textarea, button).
- **$(":text"):** Selects only text elements (input[type=text]).
- **$("li:eq(2)"):** Selects the third <li> element
- **$("li:eq(4)"):** Selects the fifth <li> element
- **$("li:lt(2)"):** Selects all elements matched by <li> element before the third one; in other words, the first two <li> elements.
- **$("p:lt(3)"):** selects all elements matched by <p> elements before the fourth one; in other words the first three <p> elements.
- **$("li:gt(1)"):** Selects all elements matched by <li> after the second one.
- **$("p:gt(2)"):** Selects all elements matched by <p> after the third one.
- **$("div/p"):** Selects all elements matched by <p> that are children of an element matched by .
- **$("div//code"):** Selects all elements matched by <code>that are descendants of an element matched by .
- **$("//p//a"):** Selects all elements matched by <a> that are descendants of an element matched by <p>

- **$("li:first-child"):** Selects all elements matched by <li> that are the first child of their parent.
- **$("li:last-child"):** Selects all elements matched by <li> that are the last child of their parent.
- **$(":parent"):** Selects all elements that are the parent of another element, including text.
- **$("li:contains(second)"):** Selects all elements matched by <li> that contain the text second.

You can use all the above selectors with any HTML/XML element in generic way. For example if selector **$("li:first")** works for <li> element then **$("p:first")** would also work for <p> element.

Some of the most basic components we can manipulate when it comes to DOM elements are the properties and attributes assigned to those elements.

Most of these attributes are available through JavaScript as DOM node properties. Some of the more common properties are:

- className
- tagName
- id
- href
- title
- rel
- src

Consider the following HTML markup for an image element:

```
<img id="myImage" src="image.gif" alt="An image"
class="someClass" title="This is an image"/>
```

In this element's markup, the tag name is img, and the markup for id, src, alt, class, and title represents the element's attributes, each of which consists of a name and a value.

jQuery gives us the means to easily manipulate an element's attributes and gives us access to the element so that we can also change its properties.

Get Attribute Value:

The **attr()** method can be used to either fetch the value of an attribute from the first element in the matched set or set attribute values onto all matched elements.

Example:

Following is a simple example which fetches title attribute of tag and set value with the same value:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    var title = $("em").attr("title");
    $("#divid").text(title);
  });

  </script>
</head>
<body>
  <div>
    <em title="Bold and Brave">This is first paragraph.</em>
    <p id="myid">This is second paragraph.</p>
    <div id="divid"></div>
  </div>
</body>
</html>
```

Set Attribute Value:

The **attr(name, value)** method can be used to set the named attribute onto all elements in the wrapped set using the passed value.

Example:

Following is a simple example which set **src** attribute of an image tag to a correct location:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("#myimg").attr("src", "/images/jquery.jpg");
  });

  </script>
</head>
<body>
  <div>
    <img id="myimg" src="/wongpath.jpg" alt="Sample image" />
  </div>
</body>
</html>
```

Applying Styles:

The **addClass(classes)** method can be used to apply defined style sheets onto all the matched elements. You can specify multiple classes separated by space.

Example:

Following is a simple example which set **src** attribute of an image tag to a correct location:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("em").addClass("selected");
    $("#myid").addClass("highlight");
  });

  </script>
  <style>
    .selected { color:red; }
    .highlight { background:yellow; }
 </style>
</head>
<body>
  <em title="Bold and Brave">This is first paragraph.</em>
  <p id="myid">This is second paragraph.</p>
</body>
</html>
```

Useful Attribute Methods:

Following table lists down few useful methods which you can use to manipulate attributes and properties:

Methods	Description
attr(properties)	Set a key/value object as properties to all matched elements.
attr(key, fn)	Set a single property to a computed value, on all matched elements.
removeAttr(name)	Remove an attribute from each of the matched elements.
hasClass(class)	Returns true if the specified class is present on at least one of the set of matched elements.
removeClass(class)	Removes all or the specified class(es) from the set of matched elements.

toggleClass(class)	Adds the specified class if it is not present, removes the specified class if it is present.
html( )	Get the html contents (innerHTML) of the first matched element.
html(val)	Set the html contents of every matched element.
text( )	Get the combined text contents of all matched elements.
text(val)	Set the text contents of all matched elements.
val( )	Get the input value of the first matched element.
val(val)	Set the value attribute of every matched element if it is called on <input> but if it is called on <select> with the passed <option> value then passed option would be selected, if it is called on check box or radio box then all the matching check box and radiobox would be checked.

Similar to above syntax and examples, following examples would give you understanding on using various attribute methods in different situation:

- **$("#myID").attr("custom") :** This would return value of attribute *custom* for the first element matching with ID myID.
- **$("img").attr("alt", "Sample Image"):** This sets the **alt** attribute of all the images to a new value "Sample Image".
- **$("input").attr({ value: "", title: "Please enter a value" }); :** Sets the value of all <input> elements to the empty string, as well as sets the title to the string *Please enter a value*.
- **$("a[href^=http://]").attr("target","_blank"):** Selects all links with an href attribute starting with *http://* and set its target attribute to *_blank*
- **$("a").removeAttr("target") :** This would remove *target* attribute of all the links.
- **$("form").submit(function() {$(":submit",this).attr("disabled", "disabled");}); :** This would modify the disabled attribute to the value "disabled" while clicking Submit button.
- **$("p:last").hasClass("selected"):** This return true if last <p> tag has associated class*selected*.
- **$("p").text():** Returns string that contains the combined text contents of all matched <p> elements.
- **$("p").text("Hello World"):** This would set "<I>Hello World</I>" as text content of the matching <p> elements
- **$("p").html() :** This returns the HTML content of the all matching paragraphs.
- **$("div").html("Hello World") :** This would set the HTML content of all matching to *Hello World*.
- **$("input:checkbox:checked").val() :** Get the first value from a checked checkbox

- **$("input:radio[name=bar]:checked").val():** Get the first value from a set of radio buttons
- **$("button").val("Hello") :** Sets the value attribute of every matched element <button>.
- **$("input").val("on") :** This would check all the radio or check box button whose value is "on".
- **$("select").val("Orange") :** This would select Orange option in a dropdown box with options Orange, Mango and Banana.
- **$("select").val("Orange", "Mango") :** This would select Orange and Mango options in a dropdown box with options Orange, Mango and Banana.

select elements in a document randomly as well as in sequential method.

Most of the DOM Traversal Methods do not modify the jQuery object and they are used to filter out elements from a document based on given conditions.

Find Elements by index:

Consider a simple document with the following HTML content:

```
<html>
<head>
<title>the title</title>
</head>
<body>
  <div>
  <ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
    <li>list item 6</li>
  </ul>
  </div>
</body>
</html>
```

- Above every list has its own index, and can be located directly by using **eq(index)** method as below example.
- Every child element starts its index from zero, thus, *list item 2* would be accessed by using **$("li").eq(1)** and so on.

Example:

Following is a simple example which adds the color to second list item.

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("li").eq(2).addClass("selected");
  });

  </script>
  <style>
    .selected { color:red; }
 </style>
</head>
<body>
  <div>
  <ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
    <li>list item 6</li>
  </ul>
  </div>
</body>
</html>
```

Filtering out Elements:

The **filter(selector)** method can be used to filter out all elements from the set of matched elements that do not match the specified selector(s). The *selector* can be written using any selector syntax.

Example:

Following is a simple example which applies color to the lists associated with middle class:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("li").filter(".middle").addClass("selected");
  });

  </script>
  <style>
    .selected { color:red; }
 </style>
</head>
<body>
  <div>
  <ul>
    <li class="top">list item 1</li>
    <li class="top">list item 2</li>
    <li class="middle">list item 3</li>
    <li class="middle">list item 4</li>
    <li class="bottom">list item 5</li>
    <li class="bottom">list item 6</li>
  </ul>
  </div>
</body>
</html>
```

Locating Descendent Elements

The **find(selector)** method can be used to locate all the descendent elements of a particular type of elements. The *selector* can be written using any selector syntax.

Example:

Following is an example which selects all the elements available inside different <p> elements:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("p").find("span").addClass("selected");
  });

  </script>
  <style>
    .selected { color:red; }
 </style>
</head>
<body>
  <p>This is 1st paragraph and <span>THIS IS RED</span></p>
  <p>This is 2nd paragraph and <span>THIS IS ALSO RED</span></p>
</body>
</html>
```

JQuery DOM Traversing Methods

Following table lists down useful methods which you can use to filter out various elements from a list of DOM elements:

Selector	Description
eq(index)	Reduce the set of matched elements to a single element.
filter(selector)	Removes all elements from the set of matched elements that do not match the specified selector(s).
filter(fn)	Removes all elements from the set of matched elements that do not match the specified function.

is(selector)	Checks the current selection against an expression and returns true, if at least one element of the selection fits the given selector.
map(callback)	Translate a set of elements in the jQuery object into another set of values in a jQuery array (which may, or may not contain elements).
not(selector)	Removes elements matching the specified selector from the set of matched elements.
slice(start, [end])	Selects a subset of the matched elements.

Following table lists down other useful methods which you can use to locate various elements in a DOM:

Selector	Description
add(selector)	Adds more elements, matched by the given selector, to the set of matched elements.
andSelf( )	Add the previous selection to the current selection.
children([selector])	Get a set of elements containing all of the unique immediate children of each of the matched set of elements.
closest(selector)	Get a set of elements containing the closest parent element that matches the specified selector, the starting element included.
contents( )	Find all the child nodes inside the matched elements (including text nodes), or the content document, if the element is an iframe.
end( )	Revert the most recent 'destructive' operation, changing the set of matched elements to its previous state .
find(selector)	Searches for descendent elements that match the specified selectors.
next([selector])	Get a set of elements containing the unique next siblings of each of the given set of elements.
nextAll([selector])	Find all sibling elements after the current element.

offsetParent( )	Returns a jQuery collection with the positioned parent of the first matched element.
parent([selector])	Get the direct parent of an element. If called on a set of elements, parent returns a set of their unique direct parent elements.
parents([selector])	Get a set of elements containing the unique ancestors of the matched set of elements (except for the root element).
prev([selector])	Get a set of elements containing the unique previous siblings of each of the matched set of elements.
prevAll([selector])	Find all sibling elements in front of the current element.
siblings([selector])	Get a set of elements containing all of the unique siblings of each of the matched set of elements.

The jQuery library supports nearly all of the selectors included in Cascading Style Sheet (CSS) specifications 1 through 3, as outlined on the World Wide Web Consortium's site.

Using JQuery library developers can enhance their websites without worrying about browsers and their versions as long as the browsers have JavaScript enabled.

Most of the JQuery CSS Methods do not modify the content of the jQuery object and they are used to apply CSS properties on DOM elements.

Apply CSS Properties

This is very simple to apply any CSS property using JQuery method **css(PropertyName, PropertyValue)**.

Here is the syntax for the method:

selector.css(PropertyName, PropertyValue);

Here you can pass *PropertyName* as a javascript string and based on its value, *PropertyValue* could be string or integer.

Example:

Following is an example which adds font color to the second list item.

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("li").eq(2).css("color", "red");
  });

  </script>
</head>
<body>
  <div>
  <ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
    <li>list item 6</li>
  </ul>
  </div>
</body>
</html>
```

Apply Multiple CSS Properties:

You can apply multiple CSS properties using a single JQuery method **CSS({key1:val1, key2:val2....)**. You can apply as many properties as you like in a single call.

Here is the syntax for the method:

selector.css({key1:val1, key2:val2....keyN:valN})

Here you can pass key as property and val as its value as described above.

Example:

Following is an example which adds font color as well as background color to the second list item.

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("li").eq(2).css({"color":"red",
                  "background-color":"green"});
  });

  </script>
</head>
<body>
  <div>
  <ul>
    <li>list item 1</li>
    <li>list item 2</li>
    <li>list item 3</li>
    <li>list item 4</li>
    <li>list item 5</li>
    <li>list item 6</li>
  </ul>
  </div>
</body>
</html>
```

Setting Element Width & Height

The **width(val)** and **height(val)** method can be used to set the width and hieght respectively of any element.

Example:

Following is a simple example which sets the width of first division element where as rest of the elements have width set by style sheet:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $("div:first").width(100);
    $("div:first").css("background-color", "blue");
  });

  </script>
  <style>
  div{ width:70px; height:50px; float:left; margin:5px;
    background:red; cursor:pointer; }
 </style>
</head>
<body>
 <div></div>
 <div>d</div>
 <div>d</div>
 <div>d</div>
 <div>d</div>
</body>
</html>
```

JQuery CSS Methods

Following table lists down all the methods which you can use to play with CSS properties:

Method	Description
css(name)	Return a style property on the first matched element.
css(name, value)	Set a single style property to a value on all matched elements.

css(properties)	Set a key/value object as style properties to all matched elements.
height(val)	Set the CSS height of every matched element.
height( )	Get the current computed, pixel, height of the first matched element.
innerHeight( )	Gets the inner height (excludes the border and includes the padding) for the first matched element.
innerWidth( )	Gets the inner width (excludes the border and includes the padding) for the first matched element.
offset( )	Get the current offset of the first matched element, in pixels, relative to the document
offsetParent( )	Returns a jQuery collection with the positioned parent of the first matched element.
outerHeight([margin])	Gets the outer height (includes the border and padding by default) for the first matched element.
outerWidth([margin])	Get the outer width (includes the border and padding by default) for the first matched element.
position( )	Gets the top and left position of an element relative to its offset parent.
scrollLeft(val)	When a value is passed in, the scroll left offset is set to that value on all matched elements.
scrollLeft( )	Gets the scroll left offset of the first matched element.
scrollTop(val)	When a value is passed in, the scroll top offset is set to that value on all matched elements.
scrollTop( )	Gets the scroll top offset of the first matched element.
width(val)	Set the CSS width of every matched element.
width( )	Get the current computed, pixel, width of the first matched element.

JQuery provides methods to manipulate DOM in efficient way. You do not need to write big code to modify the value of any element's attribute or to extract HTML code from a paragraph or division.

JQuery provides methods such as .attr(), .html(), and .val() which act as getters, retrieving information from DOM elements for later use.

Content Manipulation:

The **html( )** method gets the html contents (innerHTML) of the first matched element.

Here is the syntax for the method:

selector.html( )

Example:

Following is an example which makes use of .html() and .text(val) methods. Here .html() retrieves HTML content from the object and then .text(val) method sets value of the object using passed parameter:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
     var content = $(this).html();
     $("#result").text( content );
   });

  });

  </script>
  <style>
    #division{ margin:10px;padding:12px;
            border:2px solid #666;
            width:60px;
           }
  </style>
</head>
```

```
<body>
  <p>Click on the square below:</p>
  <span id="result"> </span>
  <div id="division" style="background-color:blue;">
   This is Blue Square!!
  </div>
</body>
</html>
```

DOM Element Replacement:

You can replace a complete DOM element with the specified HTML or DOM elements. The **replaceWith(content)** method serves this purpose very well.

Here is the syntax for the method:

selector.replaceWith(content)

Here content is what you want to have instead of original element. This could be HTML or simple text.

Example:

Following is an example which would replace division element with "<h1>JQuery is Great</h1>":

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).replaceWith("<h1>JQuery is Great</h1>");
   });

  });

  </script>
```

```
  <style>
    #division{ margin:10px;padding:12px;
            border:2px solid #666;
            width:60px;
           }
  </style>
</head>
<body>
  <p>Click on the square below:</p>
  <span id="result"> </span>
  <div id="division" style="background-color:blue;">
    This is Blue Square!!
  </div>
</body>
</html>
```

Removing DOM Elements:

There may be a situation when you would like to remove one or more DOM elements from the document. JQuery provides two methods to handle the situation.

The **empty( )** method remove all child nodes from the set of matched elements where as the method **remove(expr)** method removes all matched elements from the DOM.

Here is the syntax for the method:

selector.remove([expr])

or

selector.empty( )

You can pass optional paramter *expr* to filter the set of elements to be removed.

Example:

Following is an example where elements are being removed as soon as they are clicked:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
```

```
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).remove( );
   });

  });

  </script>
  <style>
    .div{ margin:10px;padding:12px;
          border:2px solid #666;
          width:60px;
         }
  </style>
</head>
<body>
  <p>Click on any square below:</p>
  <span id="result"> </span>
  <div class="div" style="background-color:blue;"></div>
  <div class="div" style="background-color:green;"></div>
  <div class="div" style="background-color:red;"></div>
</body>
</html>
```

Inserting DOM elements:

There may be a situation when you would like to insert new one or more DOM elements in your existing document. JQuery provides various methods to insert elements at various locations.

The **after(content)** method insert content after each of the matched elements where as the method **before(content)** method inserts content before each of the matched elements.

Here is the syntax for the method:

selector.after(content)

or

selector.before(content)

Here content is what you want to insert. This could be HTML or simple text.

Example:

Following is an example where elements are being inserted just before the clicked element:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).before('<div class="div"></div>' );
   });

  });

  </script>
  <style>
    .div{ margin:10px;padding:12px;
          border:2px solid #666;
          width:60px;
        }
 </style>
</head>
<body>
  <p>Click on any square below:</p>
  <span id="result"> </span>
  <div class="div" style="background-color:blue;"></div>
  <div class="div" style="background-color:green;"></div>
  <div class="div" style="background-color:red;"></div>
</body>
</html>
```

DOM Manipulation Methods:

Following table lists down all the methods which you can use to manipulate DOM elements:

Method	Description
after(content)	Insert content after each of the matched elements.
append(content)	Append content to the inside of every matched element.

appendTo(selector)	Append all of the matched elements to another, specified, set of elements.
before(content)	Insert content before each of the matched elements.
clone(bool)	Clone matched DOM Elements, and all their event handlers, and select the clones.
clone( )	Clone matched DOM Elements and select the clones.
empty( )	Remove all child nodes from the set of matched elements.
html(val)	Set the html contents of every matched element.
html( )	Get the html contents (innerHTML) of the first matched element.
insertAfter(selector)	Insert all of the matched elements after another, specified, set of elements.
insertBefore(selector)	Insert all of the matched elements before another, specified, set of elements.
prepend(content)	Prepend content to the inside of every matched element.
prependTo(selector)	Prepend all of the matched elements to another, specified, set of elements.
remove(expr)	Removes all matched elements from the DOM.
replaceAll(selector)	Replaces the elements matched by the specified selector with the matched elements.
replaceWith(content)	Replaces all matched elements with the specified HTML or DOM elements.
text(val)	Set the text contents of all matched elements.
text( )	Get the combined text contents of all matched elements.
wrap(elem)	Wrap each matched element with the specified element.
wrap(html)	Wrap each matched element with the specified HTML content.

wrapAll(elem)	Wrap all the elements in the matched set into a single wrapper element.
wrapAll(html)	Wrap all the elements in the matched set into a single wrapper element.
wrapInner(elem)	Wrap the inner child contents of each matched element (including text nodes) with a DOM element.
wrapInner(html)	Wrap the inner child contents of each matched element (including text nodes) with an HTML structure.

Query provides methods to manipulate DOM in efficient way. You do not need to write big code to modify the value of any element's attribute or to extract HTML code from a paragraph or division.

JQuery provides methods such as .attr(), .html(), and .val() which act as getters, retrieving information from DOM elements for later use.

Content Manipulation:

The **html( )** method gets the html contents (innerHTML) of the first matched element.

Here is the syntax for the method:

selector.html( )

Example:

Following is an example which makes use of .html() and .text(val) methods. Here .html() retrieves HTML content from the object and then .text(val) method sets value of the object using passed parameter:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
```

```
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
     var content = $(this).html();
     $("#result").text( content );
   });

  });

  </script>
  <style>
    #division{ margin:10px;padding:12px;
            border:2px solid #666;
            width:60px;
           }
 </style>
</head>
<body>
  <p>Click on the square below:</p>
  <span id="result"> </span>
  <div id="division" style="background-color:blue;">
   This is Blue Square!!
  </div>
</body>
</html>
```

DOM Element Replacement:

You can replace a complete DOM element with the specified HTML or DOM elements. The **replaceWith(content)** method serves this purpose very well.

Here is the syntax for the method:

selector.replaceWith(content)

Here content is what you want to have instead of original element. This could be HTML or simple text.

Example:

Following is an example which would replace division element with "<h1>JQuery is Great</h1>":

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).replaceWith("<h1>JQuery is Great</h1>");
   });

  });

  </script>
  <style>
    #division{ margin:10px;padding:12px;
            border:2px solid #666;
            width:60px;
           }
 </style>
</head>
<body>
  <p>Click on the square below:</p>
  <span id="result"> </span>
  <div id="division" style="background-color:blue;">
   This is Blue Square!!
  </div>
</body>
</html>
```

Removing DOM Elements:

There may be a situation when you would like to remove one or more DOM elements from the document. JQuery provides two methods to handle the situation.

The **empty( )** method remove all child nodes from the set of matched elements where as the method **remove(expr)** method removes all matched elements from the DOM.

Here is the syntax for the method:

selector.remove([expr])

or

selector.empty( )

You can pass optional paramter *expr* to filter the set of elements to be removed.

Example:

Following is an example where elements are being removed as soon as they are clicked:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).remove( );
   });

  });

  </script>
  <style>
    .div{ margin:10px;padding:12px;
          border:2px solid #666;
          width:60px;
        }
 </style>
</head>
<body>
  <p>Click on any square below:</p>
  <span id="result"> </span>
  <div class="div" style="background-color:blue;"></div>
  <div class="div" style="background-color:green;"></div>
  <div class="div" style="background-color:red;"></div>
</body>
</html>
```

Inserting DOM elements:

There may be a situation when you would like to insert new one or more DOM elements in your existing document. JQuery provides various methods to insert elements at various locations.

The **after(content)** method insert content after each of the matched elements where as the method **before(content)** method inserts content before each of the matched elements.

Here is the syntax for the method:

selector.after(content)

or

selector.before(content)

Here content is what you want to insert. This could be HTML or simple text.

Example:

Following is an example where elements are being inserted just before the clicked element:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("div").click(function () {
      $(this).before('<div class="div"></div>' );
   });

  });

  </script>
  <style>
    .div{ margin:10px;padding:12px;
          border:2px solid #666;
          width:60px;
        }
 </style>
</head>
```

```
<body>
  <p>Click on any square below:</p>
  <span id="result"> </span>
  <div class="div" style="background-color:blue;"></div>
  <div class="div" style="background-color:green;"></div>
  <div class="div" style="background-color:red;"></div>
</body>
</html>
```

DOM Manipulation Methods:

Following table lists down all the methods which you can use to manipulate DOM elements:

Method	Description
after(content)	Insert content after each of the matched elements.
append(content)	Append content to the inside of every matched element.
appendTo(selector)	Append all of the matched elements to another, specified, set of elements.
before(content)	Insert content before each of the matched elements.
clone(bool)	Clone matched DOM Elements, and all their event handlers, and select the clones.
clone( )	Clone matched DOM Elements and select the clones.
empty( )	Remove all child nodes from the set of matched elements.
html(val)	Set the html contents of every matched element.
html( )	Get the html contents (innerHTML) of the first matched element.
insertAfter(selector)	Insert all of the matched elements after another, specified, set of elements.
insertBefore(selector)	Insert all of the matched elements before another, specified, set of elements.
prepend(content)	Prepend content to the inside of every matched element.

prependTo(selector)	Prepend all of the matched elements to another, specified, set of elements.
remove(expr)	Removes all matched elements from the DOM.
replaceAll(selector)	Replaces the elements matched by the specified selector with the matched elements.
replaceWith(content)	Replaces all matched elements with the specified HTML or DOM elements.
text(val)	Set the text contents of all matched elements.
text( )	Get the combined text contents of all matched elements.
wrap(elem)	Wrap each matched element with the specified element.
wrap(html)	Wrap each matched element with the specified HTML content.
wrapAll(elem)	Wrap all the elements in the matched set into a single wrapper element.
wrapAll(html)	Wrap all the elements in the matched set into a single wrapper element.
wrapInner(elem)	Wrap the inner child contents of each matched element (including text nodes) with a DOM element.
wrapInner(html)	Wrap the inner child contents of each matched element (including text nodes) with an HTML structure.

3

Handling Events with jQuery

In this chapter, we will cover:

Executing functions when a page has loaded
Binding and unbinding elements
Adding events to elements that will be created later
Submitting a form using jQuery
Checking for missing images
Creating a select/unselect all checkbox functionality
Capturing mouse movements
Creating keyboard shortcuts
Displaying user-selected text
Dragging elements on a page

Events are actions that execute some JavaScript code for producing the desired result. They can be either some sort of manipulation of a document or some internal calculations.

Since different browsers handle events differently, it takes a lot of effort to write JavaScript code that is compatible with all browsers. This chapter will help you understand event handling and explore related methods of jQuery that can make scripts compatible on different browsers. You will learn to work with the keyboard and mouse events. Advanced event handling topics like dragging and keyboard shortcuts are also discussed.

AJAX applications make extensive use of JavaScript to manipulate the content and the look and feel of web pages. Web pages should have the DOM loaded before any JavaScript code tries to perform any such modification on it.

This recipe will explain how to execute the JavaScript after the content has been loaded and the DOM is ready.

Get a copy of the latest version of the jQuery library.

1. Create a file and name it as domReady.html.
2. To run any JavaScript code only after the DOM has completely loaded, write it between the curly braces of .ready() method:

```
<script type="text/javascript">
   $(document).ready(function () {
     // code written here will run only after the DOM has loaded
   });

</script>
```

jQuery ensures that code written inside .ready() gets executed only after the DOM is fully loaded. This includes the complete document tree containing the HTML, stylesheets, and other scripts. You can, therefore, manipulate the page, attach events, and do other stuff. Note that .ready() does not wait for images to load. Images can be checked using the .load() method, which is explained in a separate recipe in this chapter.

If .ready() is not used, the jQuery code does not wait for the whole document to load. Instead it will execute as it is loaded in the browser. This can throw errors if the written code tries to manipulate any HTML or CSS that has not been loaded yet.

Passing a handler to .ready()

In the previous example code we used an anonymous function with .ready(). You can also pass a handler instead of the anonymous function. It can be done as follows:

```
<script type="text/javascript">
   $(document).ready(doSomething);
   function doSomething()
   {
     // write code here
   }


</script>
```

Another method of using .ready()

Instead of writing the code in the above mentioned format, we can also use one of the below described variations for finding out when the DOM is ready:

```
$(function ()
{
});
```

Or

```
$(doSomething);
function doSomething()
{
// DOM is ready now


}
```

Multiple .ready() methods

If there are multiple script files in your application, you can have a .ready() for each of them. jQuery will run all of these after DOM loads. An example scenario may be when you
are using some plugins on a page and each one of them has a separate .js file.

This recipe will demonstrate how you can attach events to DOM elements using the .bind() method and how to remove them using the .unbind() method.

Get a latest copy of the jQuery library to use with this recipe.

1. Create a new file, in a directory named chapter1, and name it as binding.html.
2. Write the HTML markup to create some HTML elements. Create an unordered list with the names of some countries. After that, create a select box containing names of continents as options. Finally, create a button that will be used to remove the event handler from the select box.

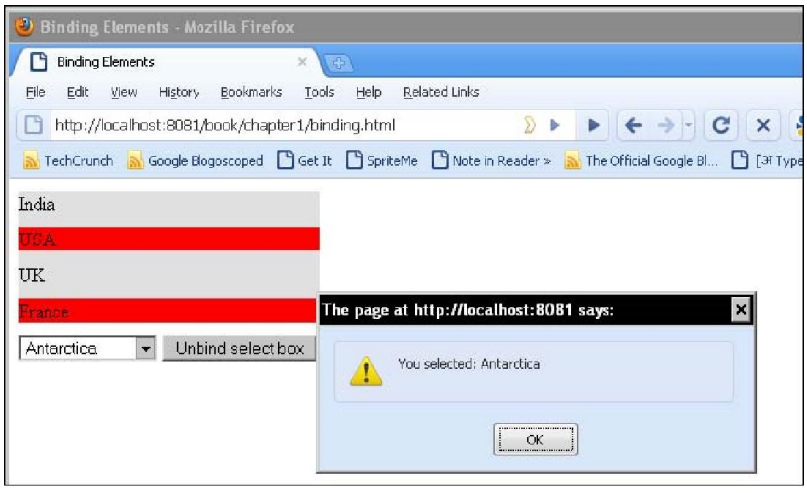
```
<html>
 <head>
  <title>Binding Elements</title>
  <style type="text/css">
   ul { background-color:#DCDCDC; list-style:none; margin:0pt;
       padding:0pt; width:250px;}
   li { cursor:pointer; margin:10px 0px;}
  </style>
 </head>
 <body>
  <ul>
   <li>India</li>
   <li>USA</li>
   <li>UK</li>
   <li>France</li>
  </ul>

  <select>
   <option value="Africa">Africa</option>
   <option value="Antarctica">Antarctica</option>
   <option value="Asia">Asia</option>
   <option value="Australia">Australia</option>
   <option value="Europe">Europe</option>
   <option value="North America">North America</option>
   <option value="South America">South America</option>
  </select>

  <input type="button" value="Unbind select box"/>
 </body>
</html>
```

3. It's time to add some jQuery magic. Attach a click event handler to list items using the .bind() method, which will set the background color of the clicked item to red. Attach the change event handler to the select box, which will display the value of the selected item. Finally, add a click handler to the button. Clicking on the button will remove the event handler from the select box.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
```

```
$(document).ready(function ()
{
 $('input:text').bind(
 {
  focus: function()
  {
  $(this).val('');
  },
  blur: function()
  {
   $(this).val('Enter some text');
  }
 });

 $('li').bind('click', function()
 {
  $(this).css('background-color', 'red');
 });

 $('select').bind('change', function()
 {
  alert('You selected: '+ $(this).val());
 });

 $('input:button').bind('click', function()
 {
  $('select').unbind('change');
 });
});
</script>
```

4. Run the binding.html file in your browser and click on some items in the list. The background color of each item clicked upon will change to red. Now select some value from the select box and you will see an alert box that displays the selected value as shown in the following screenshot:



Clicking on the **Unbind select box** button will remove the change event handler here and the selection of a value from the combo box will now do nothing.

jQuery uses the .bind() method to attach standard JavaScript events to elements. .bind() takes two parameters. The first parameter is the event type to attach. It is passed in string format, and event types such as click, change, keyup, keydown, focus, blur, and so on can be passed to it. The second parameter is the callback function, which will be executed when the event fires.

In the previous code, we used .bind() for the list items to attach a click handler. In the callback function, $(this) refers to the element that fired the event. We then use the .css() method to change the background color of the element that is clicked upon.

Similarly, we attached the change event to the select box using the .bind() method. The callback function will be called each time the value of the select box is changed.

The input button has also been attached to a click event. Clicking on the button calls the .unbind() method. This method accepts an event type name and removes that event from the element. Our example code will remove the change event from the select box. Therefore, changing the value of the select box will not display any further alerts.

Binding multiple events

Multiple events can also be attached using the .bind() method. The following code attaches two events focus and blur to a textbox. Focusing on a textbox will empty it, whereas taking the focus away from it will put some text in it.

```
$('input:text').bind(
{
 focus: function()
 {
  $(this).val('');
 },
 blur: function()
 {
  $(this).val('Enter some text');
 }


});
```

Note that this functionality was added in Version 1.4 of jQuery. So, make sure that you have the correct version before running this code.

Shortcut method for binding

Instead of using .bind(), events can be attached directly by using shortcut event names to elements. For example, $(element).click(function() { }); can be written instead of using $(element).bind('click', function() { });.

Other events can be attached similarly.

Triggering events

Events can also be triggered from the code. For this we have to pass the event name without any parameter.

```
$(element1).click(function()
{
  $(element2).keydown();
});
```

The above code will execute the keydown event of element2 when element1 is clicked.

Common event types

Here is a list of some common events that can be passed to the bind() and unbind() methods.

blur	focus
load	unload
scroll	click
dblclick	mousedown
mouseup	mousemove
mouseover	mouseout
change	select
submit	keydown
keypress	keyup

Unbinding all events from an element

If no parameter is passed to the .unbind() method, it will remove all event handlers associated with the specified element.

```
$(element).unbind();
```

The .bind() method attaches events to only those elements that exist on a page. If any new elements are created that match the criteria for the .bind() method, they will not have any event handlers.

1. Create a new file in the chapter1 directory and name it as live.html.
2. Write the HTML, which creates a button and a DIV on the page and styles them a bit.

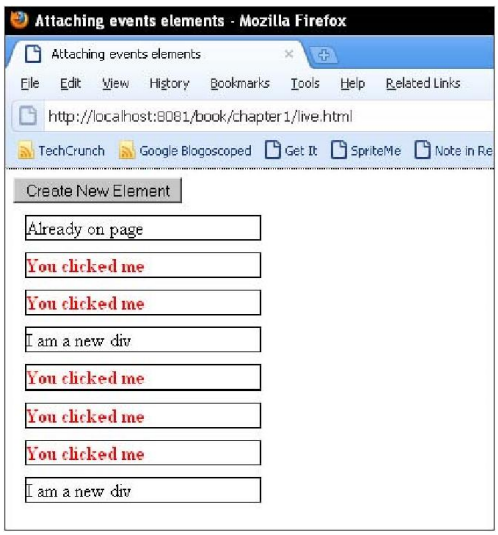
```
<html>
```

```
<head>
  <title>Attaching events elements </title>
  <style type="text/css">
  div { border: 1px solid black;cursor:pointer;width:200px;margin:10px; }
  </style>
</head>
<body>
  <input type="button" id="button" value="Create New Element"/>

  <div class="future">Already on page</div>
</body>
</html>
```

3. Time to spice things up with jQuery. Attach a click event to the button. This button will create the new DIV elements and will insert them into the page. Now attach a click event handler to the DIV using the live() method. Clicking on the DIV will change its CSS and HTML.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#button').click(function()
  {
   $('body').append('<div class="future">I am a new
     div</div>');
  });

  $('div').live('click', function()
  {
   $(this).css({'color':'red','font-weight':'bold'})
     .html('You clicked me');
   });
  });
</script>
```

4. Run the live.html file and click on the DIV. You will see that its HTML and CSS has changed. Now click on the Create New Element button a few times to create some DIV elements. Clicking on any of these DIV elements will change their appearances.
A typical screenshot after a few clicks will look similar to the following:



The input button creates the new DIV elements and appends them to the body of a document. The secret lies in the next function. We have used jQuery's live() method to attach an event on click of a DIV element. live() behaves exactly like bind() for attaching events with only one major difference. Where bind() can add events to only existing elements on a page, live() remembers the attached event for that selector and applies it to matching elements even if they are created later and then inserted into a page.

Therefore, all new DIV elements that are created as a result of clicking on the **Create New Element** button also respond to the click event handler.

Removing event handlers with die()

The die() method is similar to the unbind() method. It is used to remove event handlers that were attached using the live() method. Similar to unbind(), die() also has two variations.

If it is called with no parameters, all event handlers will be removed. Another variation accepts an event type name that will remove that particular event:

```
$(element).die();
```

The following is the code for other variations that will remove only the specified event handler.

```
$(element).die('click');
```

If an element has more than one event handler attached to it, the above code will remove only the click event handler and will leave the others intact.

> *Binding and unbinding element*s provides basic information about adding and removing events from elements.

We know that submit buttons are used in HTML forms to submit data to a server. Apart from submit buttons, JavaScript also provides a submit method that can be used to submit forms.

In this recipe, you will learn how to submit forms the jQuery way and will also learn how the form submission can be controlled using the submit button.

Get the jQuery library to use with this recipe.

1. Create a new file, name it as formSubmit.html and save it in the chapter1 directory.
2. Write the following code, which creates a form with an input button (not submit button). Add some jQuery code that will be triggered on clicking the button and will submit the form.

```
<html>
  <head>
    <title>Submitting forms</title>
  </head>
```

```
<body>
  <form id="myForm">
    <input type="button" value="Submit Form" />
  </form>
  <script type="text/javascript" src="jquery.js"></script>
  <script type="text/javascript">
    $(document).ready(function ()
    {
      $('input:button').click(function()
      {
        $('#myForm').submit();
      });
    });
  </script>
 </body>
</html>
```

3. Run the formSubmit.html file and click on the input button. It will submit the form.

In this example we attached the click event handler to the input button. The event handler function will execute when the button is clicked. On clicking the button, jQuery's submit() method is called on the form, which submits the form. All browsers have a native submit method to submit the form programmatically. jQuery has wrapped this functionality into its own submit() method.

Controlling form submission

If a form has a submit button then we can control whether to submit the form or not. In this case we will have to attach an event handler to the form. This event handler will be executed when a submit button on that particular form is clicked.

```
$('#myForm').submit(function()
{
  return false;
});
```

The above code will execute when a submit button on the form with ID myForm is clicked. If false is returned by the handler function, the form will not be submitted. This can be pretty handy for validating forms. The code for validating form values can be placed in the handler function. If values are validated, true can be returned, which will submit the form. In case the validation fails, false can be returned, which will not allow the form to be submitted.

Another option is to use preventDefault(). As the name indicates, preventDefault() prevents the default event from being executed. It is a property of the event object.

```
$('#myForm').submit(function(event)
{
  event.preventDefault()


});
```

Binding and unbinding elements explains how to add and remove events from elements.

If you are displaying some images in the browser and unfortunately some of the images are missing, the browser will either display a blank space or will display a placeholder with a cross symbol. This surely looks ugly and you would definitely want to avoid it. Wouldn't it be good if you had a method with which you could find missing images or those that failed to load?

After going through this recipe you will be able to detect missing images and replace them with an image of your choice.

Get three or four images from your computer. You will need these with this recipe. Also keep the jQuery file handy. Create another image using a program like paint with text "Could not load image" written on it. This will be the default placeholder for images that fail to load.

1. Create a new file in the chapter1 directory and name it as error.html.
2. Place a DIV in the page, which will be filled with images. Also, write some CSS to style the DIV and the images.

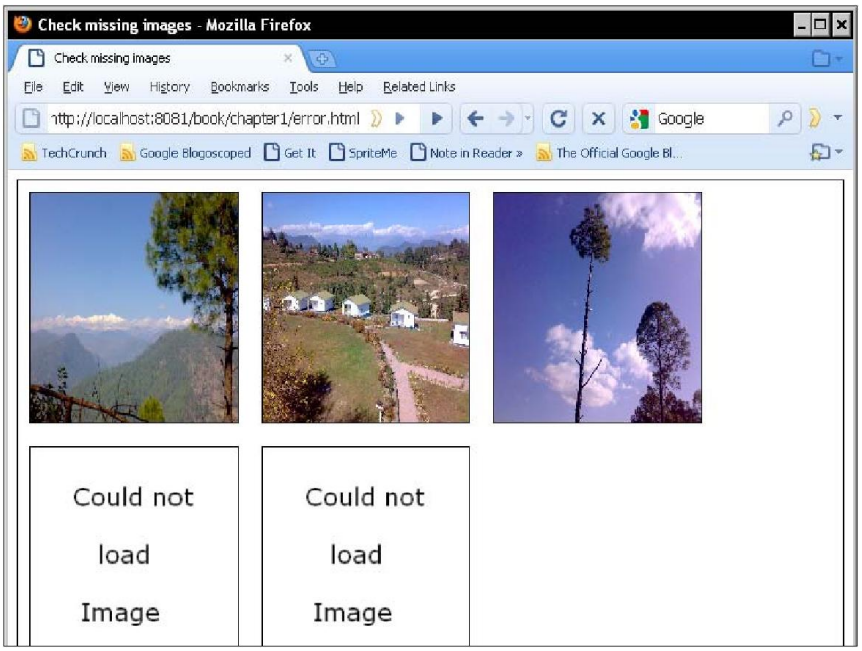
```
<html>
  <head>
    <title>Check missing images</title>
    <style type="text/css">
    div
    {
      border:1px solid black;
      float:left;
    }
    img
    {
      width:180px;
      height:200px;
      margin:10px;
    }
    </style>

  </head>
  <body>
    <div id="imageContainer"></div>
  </body>
</html>
```

3. Write the jQuery code that creates an array of image names. Intentionally put some random names of images that do not exist. Then fill the DIV by creating image tags from this array. Next, bind the error() event handler to the image elements.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
  $(document).ready(function ()
```

```
{
  var images= ['himalaya.png', 'chaukori.png', 'tree.png',
    'noSuchimage.png',  'anotheNonExistentImage.png'];
  var html = '';
  $.each(images,function(key, value)
  {
   html+= '<img src="'+value+'" />';
  });
  $('#imageContainer').html(html);

  $('img').error(function()
  {
   $(this).replaceWith('<img src="missing.png"
     alt="Could not load image">');
  });
 });
</script>
```

4. Run the error.html file in a browser. You will see that the last two images, which do not exist, have been replaced by another image that says Could not load image.



First we use jQuery's $.each() method to iterate in the array that holds image names and fills the DIV by creating image tags.

Then there is an error() event handler attached to image tags. This gets executed when the image fails to load or has a broken src attribute. The event handler for the error() method replaces the nonexistent image with another image of our choice. In our case we replace it with an image that we have created and that says **Could not load image**.

Binding and unbinding elements, which explains the basics of adding events.

This is a frequently-used feature of web applications. A group of checkboxes exists on a page, which can be controlled by a single checkbox. Clicking on the master checkbox selects all checkboxes and unchecking it deselects all.

We will create the functionality to toggle checkboxes in this recipe. We will also learn how to get values for checked elements using jQuery's selectors.

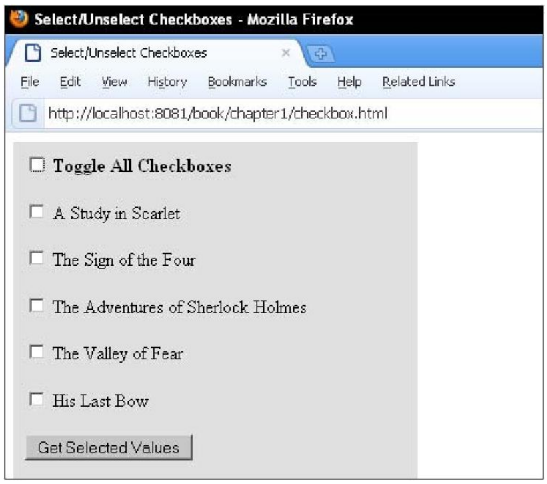
Make sure you have the jQuery library ready to be used.

1. Create a new file in the chapter1 directory and name it as checkbox.html.
2. Let us design the page first. Create an unordered list and apply some CSS to it.
   The first item in this list will be a checkbox that will work as a handle to toggle other checkboxes. Then create other items in the list: names of books each having a checkbox before it. All these checkboxes have the same class name toggle.
   Create another list item consisting of a button that will be used to display the selected books. Finally, create a last list item and assign an ID to it. We will use it to display selected book names.

```
<html>
 <head>
  <title>Select/Unselect Checkboxes</title>
  <style type="text/css">
  ul { background-color:#DCDCDC; list-style:none; margin:0pt;
       padding:0pt; width:350px;}
  li { padding:10px; }
  </style>
 </head>
 <body>
  <ul>
   <li>
    <input type="checkbox" id="handle">
    <label for="handle">
          <strong>Toggle All</strong></label>
   </li>
   <li>
    <input type="checkbox" class="toggle"/>
    <label>A Study in Scarlet</label>
   </li>
   <li>
    <input type="checkbox" class="toggle"/>
    <label>The Sign of the Four</label>
   </li>
   <li>
    <input type="checkbox" class="toggle"/>
    <label>The Adventures of Sherlock Holmes</label>
   </li>
   <li>
    <input type="checkbox" class="toggle"/>
```

```
    <label>The Valley of Fear</label>
   </li>
   <li>
    <input type="checkbox" class="toggle"/>
    <label>His Last Bow</label>
   </li>
   <li><input type="button" id="getValue"
          value="Get Selected Values"/></li>
   <li id="selected"></li>
  </ul>
 </body>
</html>
```

3. Running the checkbox.html file in browser will display the following screen:



4. To bring this page to life include the jQuery library and attach event handlers to the checkboxes. The first event handler will be attached to the first checkbox, which will take care of selecting and deselecting all other checkboxes. The second one will be attached to individual checkboxes. It will select/deselect the main handle depending on whether all checkboxes are checked or not. The last event handler is for the input button that will display the selected values beneath it.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  $('#handle').click(function(){
   if($(this).attr('checked') == true)
    $('.toggle').attr('checked', 'true');
   else
    $('.toggle').removeAttr('checked');
  });

  $('.toggle').click(function(){
   if($('.toggle:checked').length == $('.toggle').length)
    $('#handle').attr('checked', 'true');

   if($('.toggle:checked').length < $('.toggle').length)
    $('#handle').removeAttr('checked');

  });
```

```
  $('#getValue').click(function(){
    var values = '';
    if($('.toggle:checked').length)
    {
      $('.toggle:checked').each(function(){
        values+= $(this).next('label').html() + ' ,';
      });
      $('#selected').html('Selected values are: ' + values);
    }
    else
      $('#selected').html('Nothing selected');
  });
});
</script>
```

5. Now, refresh your browser and start playing with the checkboxes. Clicking on the Toggle All checkbox will select and deselect all the checkboxes alternatively. Click on the Get Selected Values button and a comma-separated list will appear below the button displaying names of all selected books.

On clicking the Toggle All checkbox we check if it is selected or not. If it is selected, we select all the other checkboxes having the class `toggle` using the class selector and set their `checked` attribute to true, which selects all the checkboxes. On the other hand, if it is not selected we remove the `checked` attribute from all checkboxes that makes all of these deselected.

We will have to take care of another issue here. If all the checkboxes are selected and any one of them is deselected, the handler checkbox should also get deselected. Similarly, if all checkboxes are selected one by one, the handler checkbox should also get checked. For this we attach another event handler to all the checkboxes having class `toggle`. The `.toggle:checked` selector selects all those elements that have class `toggle` and those which are also selected. If the length of the selected elements is equal to the total number
of checkboxes, we can conclude that all are selected and hence we select the handler checkbox too.

If the number of selected elements is less than the total number of checkboxes then we remove the `checked` attribute of the handler checkbox to deselect it.

Using selectors

In the previous example we used .toggle:checked to select all the checkboxes that have class toggle and are checked. : is a selector that is used to filter a set of elements. Listed below are examples that demonstrate how it can be used to filter elements.

```
$('div:first').click(function()

{

  //do something
});
```

The above code will select the first DIV on the page and will add a click event handler to it.

```
$(p:gt(2)').hide();
```

gt stands for greater than. It accepts a 0-based index and matches elements that have an index greater than the one specified. If a page has 5 p elements, the above example will hide p numbers 3 and 4. Remember that the index is 0-based.

You can read about all the selectors on the jQuery site at this URL:
http://api.jquery.com/category/selectors/.

jQuery can be used to determine the position of the mouse pointer on screen. This recipe explains the technique for getting the mouse pointer position on screen. You will learn how
to create a tooltip that will appear at current mouse pointer position on a particular element.

Keep the jQuery file ready to use with this recipe.

1. Open a new file in your text editor and save it in chapter1 directory as mouse.html.
2. Create a DIV with the ID tip and display set to none. This DIV will be displayed as tooltip. Create three more DIV elements and assign class hoverMe to the first and the last DIV. Write CSS styles for the DIV elements. The DIV that will be displayed as the tooltip must have position set to absolute.

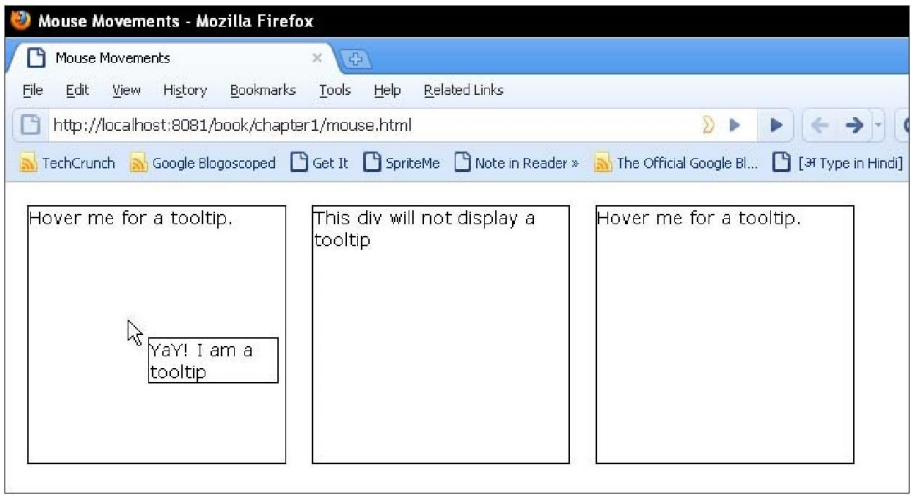
```
<html>
  <head>
    <title>Mouse Movements</title>
    <style type="text/css">
      div
      {
        border:1px solid black;
        float:left;
        width:200px;
        height:200px;
        margin:10px;
        font-family:verdana,arial;
        font-size:14px;
      }
```

```
    div#tip
    {
      position:absolute;
      width:100px;
      height:auto;
    }
  </style>
 </head>
 <body>

  <div id="tip" style="display:none;">YaY! I am a tooltip</div>

  <div class="hoverMe">Hover me for a tooltip.</div>
  <div>This div will not display a tooltip</div>
  <div class="hoverMe">Hover me for a tooltip.</div>

 </body>
</html>
```

3. Write the jQuery code that will display the tooltip when hovering over the DIV with class hoverMe. Two functions will be required for this. The first one will take care of showing and hiding the tooltip on hover with fade effect. The second function will actually set the position of tooltip and will move it as the mouse pointer moves.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('.hoverMe').hover(
  function()
  {
   $('#tip').fadeIn('slow');
  },
  function()
  {
   $('#tip').fadeOut('slow');
  });

  $('.hoverMe').mousemove(function(e)
  {
   var topPosition = e.pageY+5;
   var leftPosition = e.pageX+5;
   $('#tip').css(
   {
    'top' :  topPosition+ 'px',
    'left' : leftPosition +'px'
   });
  });
 });
</script>
```

4. Open your browser and run the mouse.html file. Hovering over the first and last DIV elements will display a tooltip with fade effect. The tooltip will also follow the mouse pointer as it moves.



We have used the hover() method on the DIV elements to show and hide the tooltip. This method attaches two event handlers to the specified element. The first event handler gets executed when the mouse pointer enters the element and the second one executes when the mouse pointer leaves that element. We have used the fadeIn() method to display the tooltip when a mouse pointer enters a DIV and the fadeout() method to hide the DIV as soon as the mouse pointer leaves it.

The most important thing now is to position the tooltip where the mouse pointer is. For this we attached an event handler mousemove on the DIV. As the name indicates, the handler function will execute when the mouse pointer is moving over the DIV. jQuery makes an event object available to the handler function, using which we can get the current mouse pointer position. The pageX property of the event gives us the cursor position relative to the left corner of the document. Similarly, the pageY property gets the mouse pointer position relative to the top of the window.

We have the mouse pointer coordinates with us now. We then assign the value of pageX and pageY to the CSS properties left and top of the tooltip DIV respectively.The value 5 has been added to each value to avoid the cursor from hiding part of the tooltip.

Keyboard navigation is common in window-based applications. This is very handy for those who prefer keyboard controls over mouse controls. Keyboard shortcuts can also be created in web applications but they are difficult to implement due to inconsistency among browsers.

We will create a simple example in this recipe that will give you the basic understanding of implementing shortcut keys. You will be able to create your own shortcut keys for use in your web applications.

Get the jQuery library to use with this recipe.

1. Create a new file named keyboard.html and save it in the chapter1 directory.
2. In the body of HTML create two DIV elements and in the <head> section write some CSS to apply styles to these DIV elements.

   <html>

```
<head>
  <title>Keyboard Shortcuts</title>
  <style type="text/css">
  div{ border : 1px solid black;float:left;height:200px;
        margin:10px; width:220px;}
  </style>
</head>
<body>
  <div>You can toggle this div using Alt+S</div>

  <div>You can toggle this div using Alt+G </div>

  <p style="clear:both;"> </p>
  <p>Press Alt+B to toggle both divs</p>
</body>
</html>
```

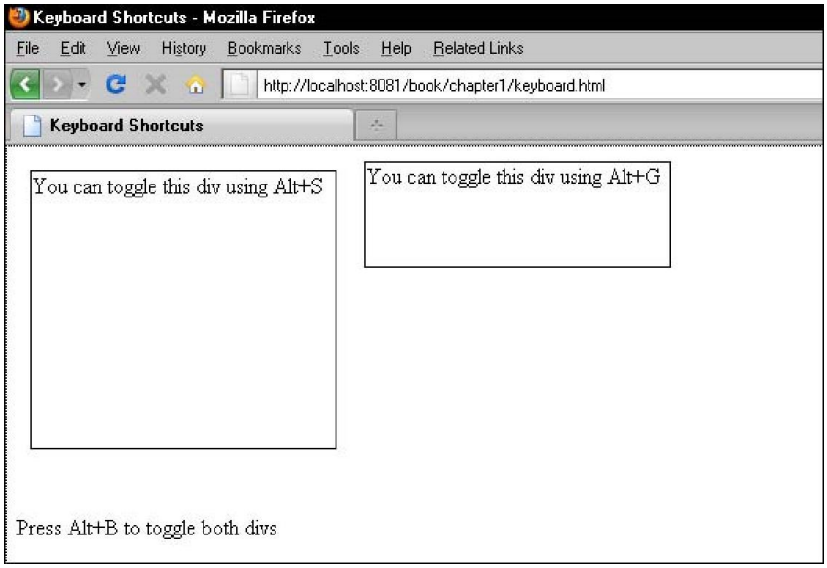
3. Write the jQuery code that will create keyboard shortcuts to toggle these DIV elements. The keydown event handler will be used to implement this behaviour. It will check for the keys that are pressed and then take actions accordingly. Three shortcuts will be created. Pressing *Alt + S* will toggle the first DIV. *Alt + G* will toggle the second DIV. Pressing *Alt + B* will toggle both the DIV elements together.

   Another handler keyup will be used to reset the required variables.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  //remember that this is a global variable
  var altPressed = false;
  $(document).keydown(function (event)
  {
   if(event.which == 18)
    altPressed = true;
   if(altPressed)
   {
     switch(event.which)
     {
       case 83:
          $('div:first').slideToggle('slow');
          return false;
          break;
       case 71:
          $('div:last').slideToggle('slow');
          return false;
          break;
       case 66:
          $('div').slideToggle('slow');
          return false;
          break;
     }
   }
  });

  $(document).keyup(function (event) {
   if(event.which == 18)
    altPressed = false;
  });
});
```

```
</script>
```

4. Open your browser and run the keyboard.html file. Try pressing the shortcuts that we have just created. You will see that the DIV elements will toggle with a slide effect.



In order to be able to create shortcut keys, first we need to find out which key was pressed. Different browsers have their own methods of determining the value of the pressed key. jQuery normalizes the way this information can be retrieved across browsers. An event object is available to handler functions. This event object has a property which that gives the code of the pressed key. *Alt* key has the value 18.

The keyboard shortcuts in this recipe use the combination of *Alt* and the other keys. We begin by declaring a global variable altPressed with the value set to false. Then there are two events attached to the page. keydown will execute when a key is in a pressed state and keyup when a key is released. Whenever *Alt* is pressed the keydown event will set its value to true. When released, it will be reset to false again by the keyup handler function.

Next comes the if statement, which will evaluate to a true value if the *Alt* key is pressed. If *Alt* is pressed and another key is pressed along with it, the switch case will check the key's value and will execute the corresponding switch case.

The value for the S key is 83. So, pressing S along with *Alt* will select the first DIV and will apply the slideToggle effect to it. Similarly, *Alt* + *G* will toggle the second DIV and *Alt* + *B* will toggle both DIVs.

Note the return of false in each case of switch statement. Returning false is necessary to override a browser's default behavior. If false is not returned, pressing the *Alt* key will activate the browser's menu.

List of common key codes

A list of key codes can be found at http://goo.gl/v2Fk

> *Binding and unbinding element*s in this chapter explains how to attach events to elements.

You must have seen the WYSIWYG (What You See Is What You Get) editors in web applications, which allow you to select some text using the mouse or keyboard and then format it (like making it bold, changing its color, and so on).

This recipe will teach you how to retrieve the text that is selected by a user and perform some basic formatting on it.

Get the jQuery library ready.

1. Create a file named textSelect.html in your chapter1 directory.
2. Create four buttons out of which the first three will be used to make the text bold, italic, and underlined respectively. Then create a textarea with some text in it. And finally, enter a paragraph that will be used to display the formatted HTML.

   The last button will get the value of textarea and will insert it in the paragraph.

```
<html>
  <head>
    <title>Manipulating user selected text</title>
    <style type="text/css">
      p { color:red;font-size:17px;width:670px;}
    </style>
  </head>
  <body>
    <input type="button" value="b" id="bold" class="button">
    <input type="button" value="i" id="italics" class="button">
    <input type="button" value="u" id="underline" class="button">
    <input type="button" id="apply" value="Apply HTML">
    <div>
    <textarea id="selectable" rows="20" cols="80">I consider that
      a man's brain originally is like a little empty attic, and
      you have to stock it with such furniture as you choose. A
      fool takes in all the lumber of every sort that he comes
      cross, so that the knowledge which might be useful to him
      gets crowded out, or at best is jumbled up with a lot of
      other things, so that he has a difficulty in laying his
      hands upon it.</textarea>
    </div>
    <p id="container"></p>
  </body>
</html>
```

3. Include the jQuery library and write the JavaScript function that will get the start and end positions of the selected text.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
function getPositions()
{
 var startPosition = endPosition = 0;
 var element = document.getElementById('selectable');
 if (document.selection)
 {
  //for Internet Explorer
  var range = document.selection.createRange();
  var drange = range.duplicate();
  drange.moveToElementText(element);
  drange.setEndPoint("EndToEnd", range);
  startPosition = drange.text.length - range.text.length;
  endPosition = startPosition  + range.text.length;
 }
 else if (window.getSelection)
 {
  //For Firefox, Chrome, Safari etc
  startPosition = element.selectionStart;
  endPosition = element.selectionEnd;

 }
 return {'start': startPosition, 'end': endPosition};
}
```

4. Next, write the code for the Apply HTML button that will simply get the text from the textarea and insert it in the paragraph.

```
$('#apply').click(function()
{
   var html = $('#container').html($('#selectable').val());
});
```

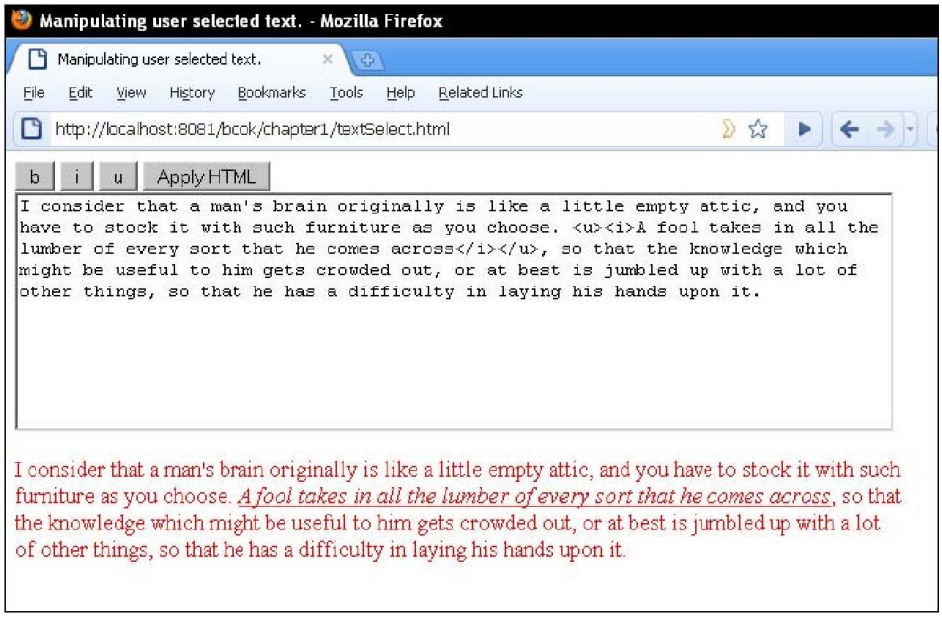
5. Let's code the first three buttons now. We will bind the click event with the three buttons. On the click of each button, the position of the selected text will be retrieved
and it will be enclosed within HTML tags depending on which button is clicked.

```
$('.button').click(function()
{
 var positions = getPositions();
 if(positions.start == positions.end)
 {
  return false;
 }
 var tag = $(this).val();
 var textOnPage = $('#selectable').val();

 var startString = textOnPage.substr(0, positions.start);

 var targetString = textOnPage.substr(positions.start,
   positions.end - positions.start);
 var formattedString = "<" + tag +">" + targetString +
  "</" +  tag +">";
 var endString = textOnPage.substr(positions.end);
 $('#selectable').text(startString + formattedString +
    endString);
```

```
});
```

6. Save the code, start your browser and point it to the file. Select some text with your mouse and click on any of the buttons. You will see that the selected text has been enclosed with the corresponding HTML tags. If you click on the second button (u), the selected text will be enclosed in <u> and </u> HTML tags.

   Now click on the **Apply HTML** button. You will be able to see the formatted text of the textarea in HTML format inside the paragraph, as seen in the following screenshot:



On click of a button, we first get the start and end positions of selected text using the getPositions() function. Determining this value is a bit complex as different browsers
have different methods for handling selections. Internet Explorer uses document.selection, which represents a subset of documents, whereas Mozilla and similar browsers use
window.getSelection.

IE has a range of objects using which we can determine what text was selected, and the start and end positions of selection in original text. First we create a range object from the selection. Then we create a clone of it using the duplicate method. After this, two functions moveToElementText() and setEndPoint() are used on the duplicated range. These methods align the values of original text and the selection.

Once this is done, we compare the values of the original and the duplicated range to find out the start position. Then we add the length of the selection to the start position, which gives
us the end position marker.

For other browsers, getting positions is relatively simple. Start and end positions of selections in textarea can be retrieved using .selectionStart and .selectionEnd properties.

Once we get both these values, we create an object in which we put both of these and return the object to the calling function.

If the values of both these positions are equal, it means that no text is selected. In this case we simply return from the function and do nothing.

Then we determine which button was clicked. The clicked button's value will be used to format the selected

text. After that, we store the value of textarea in a local variable textOnPage.

Now comes the part where the actual manipulation takes place. We break the textOnPage variable into three parts. The first part contains the string from the beginning to the starting position of the selection. The second part of the string is the actual selected text of textarea that has to be formatted. We now enclose it in HTML tags (<b>, <i>, or <u>) according to the button clicked. The third and final part is from where the selection ends to the end of the string.

To get the resulting string we can now simply concatenate these three strings and place it back into the textarea. The textarea will now have text that has the selected text enclosed in HTML tags. To verify this, click on the **Apply HTML** button. This will take the text from the textarea and insert it as HTML into the paragraph with ID container.

Short method for getting selected text

Another method can be used to get the selected text from other elements, such as , <p>, and so on. This will not give any positions but simply the selected text. Note that this method will not work for textareas for Mozilla and similar browsers but it will work in Internet Explorer for textareas as well as other controls.

Use the following function to get the selected text:

```
function getSelectedText()
{
 var selectedText = '';
 if (document.selection)
 {
 var range = document.selection.createRange();
  selectedText = range.text;
  }
  else if (window.getSelection)
  {
    selectedText = window.getSelection();
  }
return selectedText;
```

```
}
```

There are many plugins based on JavaScript, jQuery, and other libraries, which let users implement the dragging functionality. A user presses the mouse button on an element and moves it without releasing it. The element gets dragged along with the mouse pointer. The dragging stops once the mouse key is released.

After finishing this recipe, you will be able to implement a dragging feature for elements on your own. This recipe will show you how to make elements on a page draggable.

Get the jQuery library to use with this recipe.

1. Create a new file in the chapter1 directory and name it as drag.html.
2. Create some DIV elements and assign the dragMe class to customize their appearance. This class will also be used to attach event handlers to the DIV.

```
<html>
  <head>
    <title>Dragging</title>
    <style type="text/css">
      .dragMe
      {
        background-color:#8FBC8F;
        border:1px solid black;
        color: #fff;
        float:left;
        font-family:verdana,arial;
        font-size:14px;
        font-weight:bold;
        height:100px;
        margin:10px;
        text-align:center;
        width:100px;
      }
    </style>
  </head>
  <body>

    <div class="dragMe">Drag Me</div>
    <div class="dragMe">Drag Me too</div>
  </body>
</html>
```

3. In the jQuery code, declare variables that will hold the coordinates of DIV being dragged and the mouse pointer. Proceed to attach event handlers for mouse movement to elements with the dragMe class.

   We have attached two event handlers. The first is mousedown, which will execute while the mouse button is in a pressed state on the target DIV. This will get the current left and top coordinates of the DIV being dragged and the mouse pointer.
   Now bind the mousemove element to the current DIV. The dragElement function will be called when the mouse moves while its button is pressed.

   The function dragElement calculates new values for the top and left of the DIV by determining mouse movements and the DIV's current position and applies these properties to the DIV. This results in the movement of the DIV.

   Finally, bind the mouseup event to the document, which will stop the dragging after the mouse has been released.

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  var mousex = 0, mousey = 0;
  var divLeft, divTop;
  $('.dragMe').mousedown(function(e)
  {
   var offset = $(this).offset();
   divLeft = parseInt(offset.left,10);
   divTop = parseInt(offset.top,10);
   mousey = e.pageY;
   mousex = e.pageX;
   $(this).bind('mousemove',dragElement);
  });

  function dragElement(event)
  {
   var left = divLeft + (event.pageX - mousex);
   var top = divTop + (event.pageY - mousey);
   $(this).css(
   {
    'top' :  top + 'px',
    'left' : left + 'px',
    'position' : 'absolute'
```

```
    });
    return false;
  }
  $(document).mouseup(function()
  {
    $('.dragMe').unbind('mousemove');
  });

 });
</script>
```

4. Open the browser and run the drag.html file. Both DIV elements would be draggable by now. You will now be able to drag any of these DIV elements by
pressing the mouse button over them and moving them around.

Global variables mousex and mousey will be used to store the left and top positions for the mouse pointer, and the divLeft and divTop variable will store the left and top coordinates of the DIV. Then we attached two event handlers to the DIV with class dragMe. First is mousedown, which will execute when the mouse button is in a pressed state on the target DIV. In this function get the left and top positions of the DIV being dragged and store them in the divLeft and divTop variables respectively. Secondly, get the left and top values for the current mouse pointer position from the event object and save them in the mousex and mousey variables. Now when the button is pressed, bind the mousemove element to current DIV. The dragElement function will be called when the mouse pointer moves while its button is pressed.

The dragElement function now calculates the new left and top values for the DIV being dragged. To calculate the new value for left, take the left value for the DIV (divLeft) and add the difference in the mouse position to it. The difference in mouse position can be calculated by subtracting the previous left value for mouse pointer from the current left value. Similarly calculate the new value for top.

After both these values are calculated, use the css() method to apply these values to the DIV being dragged. Don't forget to set the position as absolute. Without absolute positioning the DIV will not be able to move.

Capturing mouse movements in this chapter explains the method of retrieving mouse coordinates.

Binding and unbinding elements in this chapter teaches the basics of event handling.

We have the ability to create dynamic web pages by using events. Events are actions that can be detected by your Web Application.

Following are the examples events:

- A mouse click
- A web page loading
- Taking mouse over an element
- Submitting an HTML form
- A keystroke on your keyboard
- etc.

When these events are triggered you can then use a custom function to do pretty much whatever you want with the event. These custom functions call Event Handlers.

Binding event handlers:

Using the jQuery Event Model, we can establish event handlers on DOM elements with the **bind()** method as follows:

```
$('div').bind('click', function( event ){
  alert('Hi there!');
});
```

This code will cause the division element to respond to the click event; when a user clicks inside this division thereafter, the alert will be shown.

The full syntax of the bind() command is as follows:

selector.bind(eventType[, eventData], handler)

Following is the description of the parameters:

- **eventType:** A string containing a JavaScript event type, such as click or submit. Refer to the next section for a complete list of event types.
- **eventData:** This is optional parameter is a map of data that will be passed to the event handler.
- **handler:** A function to execute each time the event is triggered.

Removing event handlers:

Typically, once an event handler is established, it remains in effect for the remainder of the life of the page. There may be a need when you would like to remove event handler.

jQuery provides the **unbind()** command to remove an exiting event handler. The syntax of unbind() is as follows:

selector.unbind(eventType, handler)

or

selector.unbind(eventType)

Following is the description of the parameters:

- **eventType:** A string containing a JavaScript event type, such as click or submit. Refer to the next section for a complete list of event types.
- **handler:** If provided, identifies the specific listener that.s to be removed.

Event Types:

The following are cross platform and recommended event types which you can bind using JQuery:

Event Type	Description
blur	Occurs when the element loses focus
change	Occurs when the element changes
click	Occurs when a mouse click
dblclick	Occurs when a mouse double-click
error	Occurs when there is an error in loading or unloading etc.
focus	Occurs when the element gets focus
keydown	Occurs when key is pressed
keypress	Occurs when key is pressed and released
keyup	Occurs when key is released
load	Occurs when document is loaded

mousedown	Occurs when mouse button is pressed
mouseenter	Occurs when mouse enters in an element region
mouseleave	Occurs when mouse leaves an element region
mousemove	Occurs when mouse pointer moves
mouseout	Occurs when mouse pointer moves out of an element
mouseover	Occurs when mouse pointer moves over an element
mouseup	Occurs when mouse button is released
resize	Occurs when window is resized
scroll	Occurs when window is scrolled
select	Occurs when a text is selected
submit	Occurs when form is submitted
unload	Occurs when documents is unloaded

The Event Object:

The callback function takes a single parameter; when the handler is called the JavaScript event object will be passed through it.

The event object is often unneccessary and the parameter is omitted, as sufficient context is usually available when the handler is bound to know exactly what needs to be done when the handler is triggered, however there are certail attributes which you would need to be accessed.

The Event Attributes:

The following event properties/attributes are available and safe to access in a platform independent manner:

Property	Description
altKey	Set to true if the Alt key was pressed when the event was triggered, false if not. The Alt key is labeled Option on most Mac keyboards.
ctrlKey	Set to true if the Ctrl key was pressed when the event was triggered, false if not.
data	The value, if any, passed as the second parameter to the bind()

	command when the handler was established.
keyCode	For keyup and keydown events, this returns the key that was pressed.
metaKey	Set to true if the Meta key was pressed when the event was triggered, false if not. The Meta key is the Ctrl key on PCs and the Command key on Macs.
pageX	For mouse events, specifies the horizontal coordinate of the event relative from the page origin.
pageY	For mouse events, specifies the vertical coordinate of the event relative from the page origin.
relatedTarget	For some mouse events, identifies the element that the cursor left or entered when the event was triggered.
screenX	For mouse events, specifies the horizontal coordinate of the event relative from the screen origin.
screenY	For mouse events, specifies the vertical coordinate of the event relative from the screen origin.
shiftKey	Set to true if the Shift key was pressed when the event was triggered, false if not.
target	Identifies the element for which the event was triggered.
timeStamp	The timestamp (in milliseconds) when the event was created.
type	For all events, specifies the type of event that was triggered (for example, click).
which	For keyboard events, specifies the numeric code for the key that caused the event, and for mouse events, specifies which button was pressed (1 for left, 2 for middle, 3 for right)

The Event Methods:

There is a list of methods which can be called on an Event Object:

Method	Description
preventDefault()	Prevents the browser from executing the default action.
isDefaultPrevented()	Returns whether event.preventDefault() was ever called on this event object.
stopPropagation()	Stops the bubbling of an event to parent elements, preventing any parent handlers from being notified of the event.
isPropagationStopped()	Returns whether event.stopPropagation() was ever called on this event object.
stopImmediatePropagation()	Stops the rest of the handlers from being executed.
isImmediatePropagationStopped()	Returns whether event.stopImmediatePropagation() was ever called on this event object.

Event Manipulation Methods:

Following table lists down important event-related methods:

Method	Description
bind(type, [data], fn)	Binds a handler to one or more events (like click) for each matched element. Can also bind custom events.
die(type, fn)	This does the opposite of live, it removes a bound live event.
hover(over, out)	Simulates hovering for example moving the mouse on, and off, an object.
live(type, fn)	Binds a handler to an event (like click) for all current - and future - matched element. Can also bind custom events.
one(type, [data], fn)	Binds a handler to one or more events to be executed once for each matched element.
ready(fn)	Binds a function to be executed whenever the DOM is ready

	to be traversed and manipulated.
toggle(fn, fn2, fn3,...)	Toggle among two or more function calls every other click.
trigger(event, [data])	Trigger an event on every matched element.
triggerHandler(event, [data])	Triggers all bound event handlers on an element .
unbind([type], [fn])	This does the opposite of bind, it removes bound events from each of the matched elements.

Event Helper Methods:

jQuery also provides a set of event helper functions which can be used either to trigger an event to bind any event types mentioned above.

Trigger Methods:

Following is an example which would triggers the blur event on all paragraphs:

```
$("p").blur();
```

Binding Methods:

Following is an example which would bind a **click** event on all the :

```
$("div").click( function () {
  // do something here
});
```

Here is a complete list of all the support methods provided by jQuery:

Method	Description
blur( )	Triggers the blur event of each matched element.
blur(fn)	Bind a function to the blur event of each matched element.
change( )	Triggers the change event of each matched element.

change(fn)	Binds a function to the change event of each matched element.
click( )	Triggers the click event of each matched element.
click(fn)	Binds a function to the click event of each matched element.
dblclick( )	Triggers the dblclick event of each matched element.
dblclick(fn)	Binds a function to the dblclick event of each matched element.
error( )	Triggers the error event of each matched element.
error(fn)	Binds a function to the error event of each matched element.
focus( )	Triggers the focus event of each matched element.
focus(fn)	Binds a function to the focus event of each matched element.
keydown( )	Triggers the keydown event of each matched element.
keydown(fn)	Bind a function to the keydown event of each matched element.
keypress( )	Triggers the keypress event of each matched element.
keypress(fn)	Binds a function to the keypress event of each matched element.
keyup( )	Triggers the keyup event of each matched element.
keyup(fn)	Bind a function to the keyup event of each matched element.
load(fn)	Binds a function to the load event of each matched element.
mousedown(fn)	Binds a function to the mousedown event of each matched element.
mouseenter(fn)	Bind a function to the mouseenter event of each matched element.

mouseleave(fn)	Bind a function to the mouseleave event of each matched element.
mousemove(fn)	Bind a function to the mousemove event of each matched element.
mouseout(fn)	Bind a function to the mouseout event of each matched element.
mouseover(fn)	Bind a function to the mouseover event of each matched element.
mouseup(fn)	Bind a function to the mouseup event of each matched element.
resize(fn)	Bind a function to the resize event of each matched element.
scroll(fn)	Bind a function to the scroll event of each matched element.
select( )	Trigger the select event of each matched element.
select(fn)	Bind a function to the select event of each matched element.
submit( )	Trigger the submit event of each matched element.
submit(fn)	Bind a function to the submit event of each matched element.
unload(fn)	Binds a function to the unload event of each matched element.

JAX is an acronym standing for Asynchronous JavaScript and XML and this technology help us to load data from the server without a browser page refresh.

JQuery is a great tool which provides a rich set of AJAX methods to develope next generation web application.

Loading simple data

This is very easy to load any static or dynamic data using JQuery AJAX. JQuery provides **load()** method to do the job:

Syntax:

Here is the simple syntax for **load()** method:

[selector].load(URL, [data], [callback]);

Here is the description of all the parameters:

- **URL:** The URL of the server-side resource to which the request is sent. It could be a CGI, ASP, JSP, or PHP script which generates data dynamically or out of a database.
- **data:** This optional parameter represents an object whose properties are serialized into properly encoded parameters to be passed to the request. If specified, the request is made using the **POST** method. If omitted, the **GET** method is used.
- **callback:** A callback function invoked after the response data has been loaded into the elements of the matched set. The first parameter passed to this function is the response text recieved from the server and second parameter is the status code.

Example:

Consider the following HTML file with a small JQuery coding:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">
  $(document).ready(function() {
    $("#driver").click(function(event){
       $('#stage').load('/jquery/result.html');
    });
  });
  </script>
</head>
<body>
```

```
  <p>Click on the button to load result.html file:</p>
  <div id="stage" style="background-color:blue;">
       STAGE
  </div>
  <input type="button" id="driver" value="Load Data" />
</body>
</html>
```

Here **load()** initiates an Ajax request to the specified URL **/jquery/result.html** file. After loading this file, all the content would be populated inside tagged with ID *stage*. Assuming, our /jquery/result.html file has just one HTML line:

```
  <h1>THIS IS RESULT...</h1>
```

When you click the given button, then result.html file gets loaded.

Getting JSON data:

There would be a situation when server would return JSON string against your request. JQuery utility function **getJSON()** parses the returned JSON string and makes the resulting string available to the callback function as first parameter to take further action.

Syntax:

Here is the simple syntax for **getJSON()** method:

```
[selector].getJSON( URL, [data], [callback] );
```

Here is the description of all the parameters:

- **URL:** The URL of the server-side resource contacted via the GET method.
- **data:** An object whose properties serve as the name/value pairs used to construct a query string to be appended to the URL, or a preformatted and encoded query string.
- **callback:** A function invoked when the request completes. The data value resulting from digesting the response body as a JSON string is passed as the first parameter to this callback, and the status as the second.

Example:

Consider the following HTML file with a small JQuery coding:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">
  $(document).ready(function() {
    $("#driver").click(function(event){
      $.getJSON('/jquery/result.json', function(jd) {
        $('#stage').html('<p> Name: ' + jd.name + '</p>');
        $('#stage').append('<p>Age : ' + jd.age+ '</p>');
        $('#stage').append('<p> Sex: ' + jd.sex+ '</p>');
      });
    });
  });
  </script>
</head>
<body>
  <p>Click on the button to load result.html file:</p>
  <div id="stage" style="background-color:blue;">
      STAGE
  </div>
  <input type="button" id="driver" value="Load Data" />
</body>
</html>
```

Here JQuery utility method **getJSON()** initiates an Ajax request to the specified URL **/jquery/result.json** file. After loading this file, all the content would be passed to the callback function which finally would be populated inside tagged with ID *stage*. Assuming, our /jquery/result.json file has following json formatted content:

```
{
"name": "Zara Ali",
"age" : "67",
"sex": "female"
}
```

When you click the given button, then result.json file gets loaded.

Passing data to the Server:

Many times you collect input from the user and you pass that input to the server for further processing. JQuery AJAX made it easy enough to pass collected data to the server using **data** parameter of any available Ajax method.

Example:

This example demonstrate how can pass user input to a web server script which would send the same result back and we would print it:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">
  $(document).ready(function() {
    $("#driver").click(function(event){
       var name = $("#name").val();
       $("#stage").load('/jquery/result.php', {"name":name} );
    });
  });
  </script>
</head>
<body>
  <p>Enter your name and click on the button:</p>
  <input type="input" id="name" size="40" /><br />
  <div id="stage" style="background-color:blue;">
       STAGE
  </div>
  <input type="button" id="driver" value="Show Result" />
</body>
</html>
```

Here is the code written in **result.php** script:

```
<?php
if( $_REQUEST["name"] )
{
  $name = $_REQUEST['name'];
  echo "Welcome ". $name;
}
?>
```

Now you can enter any text in the given input box and then click "Show Result" button to see what you have entered in the input box.

JQuery AJAX Methods:

You have seen basic concept of AJAX using JQuery. Following table lists down all important JQuery AJAX methods which you can use based your programming need:

Methods and Description
jQuery.ajax(options) Load a remote page using an HTTP request.
jQuery.ajaxSetup(options) Setup global settings for AJAX requests.
jQuery.get(url, [data], [callback], [type]) Load a remote page using an HTTP GET request.
jQuery.getJSON(url, [data], [callback]) Load JSON data using an HTTP GET request.
jQuery.getScript(url, [callback]) Loads and executes a JavaScript file using an HTTP GET request.
jQuery.post(url, [data], [callback], [type]) Load a remote page using an HTTP POST request.
load(url, [data], [callback]) Load HTML from a remote file and inject it into the DOM.
serialize( ) Serializes a set of input elements into a string of data.
serializeArray( ) Serializes all forms and form elements like the .serialize() method but returns a JSON data structure for you to work with.

JQuery AJAX Events:

You can call various JQuery methods during the life cycle of AJAX call progress. Based on different events/stages following methods are available:

You can go through all the AJAX Events.

Methods and Description
ajaxComplete(callback) Attach a function to be executed whenever an AJAX request completes.
ajaxStart(callback) Attach a function to be executed whenever an AJAX request begins and there is none already active.
ajaxError(callback) Attach a function to be executed whenever an AJAX request fails.
ajaxSend(callback) Attach a function to be executed before an AJAX request is sent.
ajaxStop(callback) Attach a function to be executed whenever all AJAX requests have ended.
ajaxSuccess(callback) Attach a function to be executed whenever an AJAX request completes successfully.

jQuery provides a trivially simple interface for doing various kind of amazing effects. jQuery methods allow us to quickly apply commonly used effects with a minimum configuration.

This tutorial covers all the important jQuery methods to create visual effects.

Showing and Hiding elements:

The commands for showing and hiding elements are pretty much what we would expect: **show()** to show the elements in a wrapped set and **hide()** to hide them.

Syntax:

Here is the simple syntax for **show()** method:

[selector].show(speed, [callback]);

Here is the description of all the parameters:

- **speed:** A string representing one of the three predefined speeds ("slow", "normal", or "fast") or the number of milliseconds to run the animation (e.g. 1000).
- **callback:** This optional parameter represents a function to be executed whenever the animation completes; executes once for each element animated against.

Following is the simple syntax for **hide()** method:

[selector].hide(speed, [callback]);

Here is the description of all the parameters:

- **speed:** A string representing one of the three predefined speeds ("slow", "normal", or "fast") or the number of milliseconds to run the animation (e.g. 1000).
- **callback:** This optional parameter represents a function to be executed whenever the animation completes; executes once for each element animated against.

Example:

Consider the following HTML file with a small JQuery coding:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {

    $("#show").click(function () {
      $(".mydiv").show( 1000 );
    });

    $("#hide").click(function () {
      $(".mydiv").hide( 1000 );
    });

  });

  </script>
  <style>
  .mydiv{ margin:10px;padding:12px;
    border:2px solid #666;
    width:100px;
    height:100px;
   }
 </style>
</head>
<body>
  <div class="mydiv">
    This is  SQUAR
  </div>

  <input id="hide" type="button" value="Hide" />
  <input id="show" type="button" value="Show" />

</body>
</html>
```

Toggling the elements:

jQuery provides methods to toggle the display state of elements between revealed or hidden. If the element is initially displayed, it will be hidden; if hidden, it will be shown.

Syntax:

Here is the simple syntax for one of the **toggle()** methods:

[selector]..toggle([speed][, callback]);

Here is the description of all the parameters:

- **speed:** A string representing one of the three predefined speeds ("slow", "normal", or "fast") or the number of milliseconds to run the animation (e.g. 1000).
- **callback:** This optional parameter represents a function to be executed whenever the animation completes; executes once for each element animated against.

Example:

We can animate any element, such as a simple containing an image:

```
<html>
<head>
<title>the title</title>
  <script type="text/javascript"
  src="/jquery/jquery-1.3.2.min.js"></script>
  <script type="text/javascript" language="javascript">

  $(document).ready(function() {
    $(".clickme").click(function(event){
      $(".target").toggle('slow', function(){
        $(".log").text('Transition Complete');
      });
    });

  });
  </script>
  <style>
  .clickme{ margin:10px;padding:12px;
    border:2px solid #666;
```

```
    width:100px;
    height:50px;
   }
  </style>
</head>
<body>
  <div class="content">
    <div class="clickme">Click Me</div>
    <div class="target">
      <img src="/images/jquery.jpg" alt="jQuery" />
    </div>
    <div class="log"></div>
</body>
</html>
```

JQuery Effect Methods:

You have seen basic concept of jQuery Effects. Following table lists down all the important methods to create different kind of effects:

Methods and Description
animate(params, [duration, easing, callback]) A function for making custom animations.
fadeIn(speed, [callback]) Fade in all matched elements by adjusting their opacity and firing an optional callback after completion.
fadeOut(speed, [callback]) Fade out all matched elements by adjusting their opacity to 0, then setting display to "none" and firing an optional callback after completion.
fadeTo(speed, opacity, callback) Fade the opacity of all matched elements to a specified opacity and firing an optional callback after completion.
hide( ) Hides each of the set of matched elements if they are shown.
hide(speed, [callback]) Hide all matched elements using a graceful animation and firing an optional callback after completion.

show( ) Displays each of the set of matched elements if they are hidden.
show(speed, [callback]) Show all matched elements using a graceful animation and firing an optional callback after completion.
slideDown(speed, [callback]) Reveal all matched elements by adjusting their height and firing an optional callback after completion.
slideToggle(speed, [callback]) Toggle the visibility of all matched elements by adjusting their height and firing an optional callback after completion.
slideUp(speed, [callback]) Hide all matched elements by adjusting their height and firing an optional callback after completion.
stop([clearQueue, gotoEnd]) Stops all the currently running animations on all the specified elements.
toggle( ) Toggle displaying each of the set of matched elements.
toggle(speed, [callback]) Toggle displaying each of the set of matched elements using a graceful animation and firing an optional callback after completion.
toggle(switch) Toggle displaying each of the set of matched elements based upon the switch (true shows all elements, false hides all elements).
jQuery.fx.off Globally disable all animations.

UI Library Based Effects:

To use these effects you would have to download jQuery UI Library **jquery-ui-1.7.2.custom.min.js** or latest version of this UI library from jQuery UI Library.

After extracting jquery-ui-1.7.2.custom.min.js file from the download, you would include this file in similar way as you include core jQuery Library file.

Methods and Description
Blind Blinds the element away or shows it by blinding it in.
Bounce Bounces the element vertically or horizontally n-times.
Clip Clips the element on or off, vertically or horizontally.
Drop Drops the element away or shows it by dropping it in.
Explode Explodes the element into multiple pieces.
Fold Folds the element like a piece of paper.
Highlight Highlights the background with a defined color.
Puff Scale and fade out animations create the puff effect.
Pulsate Pulsates the opacity of the element multiple times.
Scale Shrink or grow an element by a percentage factor.
Shake Shakes the element vertically or horizontally n-times.
Size Resize an element to a specified width and height.
Slide Slides the element out of the viewport.
Transfer Transfers the outline of an element to another.

4

Combining PHP and jQuery

In this chapter, we will cover:

Fetching data from PHP using jQuery
Creating a query string automatically for all form elements
Detecting an AJAX request in PHP
Sending data to PHP
Aborting AJAX requests
Creating an empty page and loading it in parts
Handling errors in AJAX requests
Preventing a browser from caching AJAX request
Loading JavaScript on demand to reduce page load time

You surely know how typical web applications work. You enter a URL in your browser and the browser loads that page for you. If you are required to submit a form, you will fill it and the browser sends the filled data to the server side for processing. During this time you wait for the entire page to load. If you are on a slow connection, the wait is even longer.

Let me describe another typical scenario, a web page has two select boxes. The first select box asks you to select the name of a country. You make your selection and the whole page loads to populate the second select box with the names of the cities in that country. If by mistake you made a wrong selection, fixing your mistake means another page load. Irritating isn't it?

The point I am trying to make here is: why load the complete page every time? Why can't you just select the country name and using some magic in the background be provided with the city list without loading the complete page? Maybe you can fill some other fields if the request is taking longer.

This is where AJAX fits. AJAX is short for Asynchronous JavaScript and XML. AJAX is a technique through which client-side scripts can interact with the server-side scripts using standard HTTP protocols. Data can be moved back and forth between a client and a server script without full page reloads.

Let's find out the meaning of AJAX word by word.

Asynchronous: Asynchronous means that requests are made in the background eliminating the need for a full page load. They can also be sent in parallel, and in
the meantime the user can continue interacting with other elements on the page. Users do not have to wait for AJAX requests to complete. Remember the previous country-city example? Yes it can be done.

JavaScript: JavaScript means that the request to the server originates from JavaScript. Browsers have their own implementation of what is called an XMLHttpRequest object. It is not a standard but different browsers have
their own implementation for it.

XML: AJAX requests can be made to any platform be it a PHP page or a Java page. Therefore, to exchange any data between a client and server, there arises the need for a common format that can be understood by both JavaScript and server-side language. One such format is XML. Data can be transferred between both client
and server using XML format.

The XML in AJAX does not necessarily mean XML only. Data can be exchanged in other formats as well. It can be your custom format, text, HTML, or JSON too. Most common formats today are HTML and JSON.

Since the XMLHttpRequest implementation of browsers vary, jQuery has wrapped this functionality providing us with an array of cross-browser methods to work with AJAX requests.

In this chapter, you will get to know multiple AJAX methods of jQuery to transfer data between JavaScript and PHP. You will learn to create AJAX requests, send data to the PHP script, and perform actions on the received data.

We will also go through error handling mechanisms provided by jQuery.

In this chapter, we will primarily work with HTML or text response. Since JSON and XML are topics that need to be looked upon in detail, we will discuss both of these in separate chapters.

In all the recipes, we will add the jQuery file and other jQuery code just before the body tag closes and not in the head section as you might have seen so far. Placing the files in the head section blocks the rendering of a page until all JavaScript files have been loaded. By putting them at the end of page, the HTML will be rendered without the browser blocking anything and DOM will be ready. After the page is loaded, we can then add the JavaScript or jQuery files. This will make your pages faster.

This recipe will teach you the usage of jQuery's get method to retrieve data from a PHP script. We will see a simple example where data will be fetched from the server using the get method based on user selection from a form.

Create a directory named chapter2 in your web root. Put the jQuery library file in this directory. Now create another folder inside and name it as Recipe1. As a recipe can have
more than one file, it is better to keep them separate.

1. Create a file index.html in the Recipe1 folder. Write the HTML code in it that will create a combo box with some options.

```
<html>
 <head>
  <title>jQuery.get()</title>
  <style type="text/css">
   ul{border:1px solid black; list-style:none;
      margin:0pt;padding:0pt;float:left;
      font-family:Verdana, Arial, Helvetica, sans-serif;
      font-size:12px;width:300px;}
   li{padding:10px 5px;border-bottom:1px solid black;}
  </style>
 </head>
 <body>
  <form>
   <p>
    Show list of:
    <select id="choice">
     <option value="">select</option>
     <option value="good">Good Guys</option>
     <option value="bad">Bad Guys</option>
    </select>
   </p>
   <p id="result"></p>
  </form>
 </body>
</html>
```

2. Just before the body tag closes, include the jQuery file and write the code that will attach a change event handler on the combo box. The handler function will get the selected value from the combo box and will send an AJAX request to PHP using the get method. On successful completion of the request, the response HTML will be inserted into a paragraph present on the page.

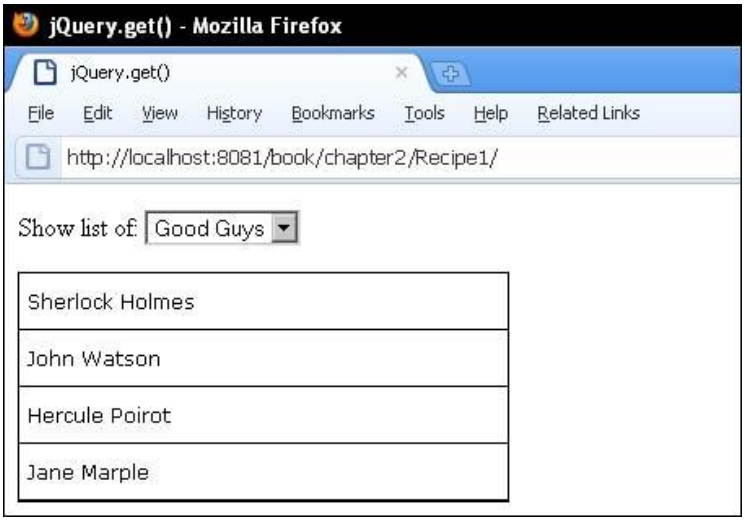
```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#choice').change(function()
  {
   if($(this).val() != '')
   {
    $.get(
     'data.php',
     { what: $(this).val() },
     function(data)
     {
      $('#result').html(data);
     });
   }
  });
 });
</script>
```

3. Let's code some PHP and respond to the AJAX request. Create a file called data.php in the same directory. Write the code that will determine the parameter received from the AJAX request. Depending on that

parameter, PHP will create some HTML using an array and will echo that HTML back to the browser.

```
<?php
 if($_GET['what'] == 'good')
 {
  $names = array('Sherlock Holmes', 'John Watson',
              'Hercule Poirot', 'Jane Marple');
  echo getHTML($names);
 }
 else if($_GET['what'] == 'bad')
 {
  $names = array('Professor Moriarty', 'Sebastian Moran',
          'Charles Milverton', 'Von Bork', 'Count Sylvius');
  echo getHTML($names);
 }
 function getHTML($names)
 {
  $strResult = '<ul>';
  for($i=0; $i<count($names); $i++)
  {
   $strResult.= '<li>'.$names[$i].'</li>';
  }
  $strResult.= '</ul>';

  return $strResult;
 }
?>
```

4. Run the index.html file in your browser and select a value from the combo box. jQuery will send the AJAX request, which will get the formatted HTML from PHP and will display it in the browser.



When a value is selected from the combo box, the corresponding event handler executes. After validating that the selected value is not blank, we send an AJAX request using the $.get() method of jQuery.

This method sends an HTTP GET request to the PHP script. $.get() accepts four parameters, which are described below. All parameters except the first one are optional:

URL: This is the file name on the server where the request will be sent. It can be the name of either a PHP file or an HTML page.

Data: This parameter defines what data will be sent to the server. It can be either in the form of a query string or a set of key-value pairs.

Handler function: This handler function is executed when the request is successful, that is, when the server roundtrip is complete.

Data type: It can be HTML, XML, JSON, or script. If none is provided, jQuery tries to make a guess itself.

Note that the URL should be from the same domain on which the application is currently running. This is because browsers do not allow cross-domain AJAX requests due to security reasons.

For example, if the page that is sending the request is http://abc.com/, it can send AJAX requests only to files located on http://abc.com/ or on its subdomains. Sending a request to other domains like http://sometothersite.com/ is not allowed.

We specified the URL as data.php. We sent a key what and set its value to a selected value of the combo box. Finally the callback function was defined. Since the method is GET, the data that will be sent to the server will be appended to the URL.

Now the request is fired and reaches the PHP file data.php. Since it is a GET request, PHP's Superglobal array $_GET will be populated with the received data. Depending on the value of key what (which can be either good or bad), PHP creates an array $names that is passed to the getHTML() function. This function creates an unordered list using the names from the array and returns it to the browser.

Note the use of echo here. echo is used to output strings on a page. In this case the page has been called through an AJAX request. Hence, the result is sent back to the function that called it. jQuery receives the response and this is available to us as a parameter of the success event handler. We insert the received HTML in a <p> element with the ID result.

Sending data to PHP later in this chapter

Creating an empty page and loading it in parts

Create a new folder Recipe2 inside the chapter2 directory. Now create a file index.html in the newly created directory.

1. Open the index.html file for editing and create a form with some HTML elements, such as textboxes, radio buttons, and check boxes.

```
<html>
<head>
```

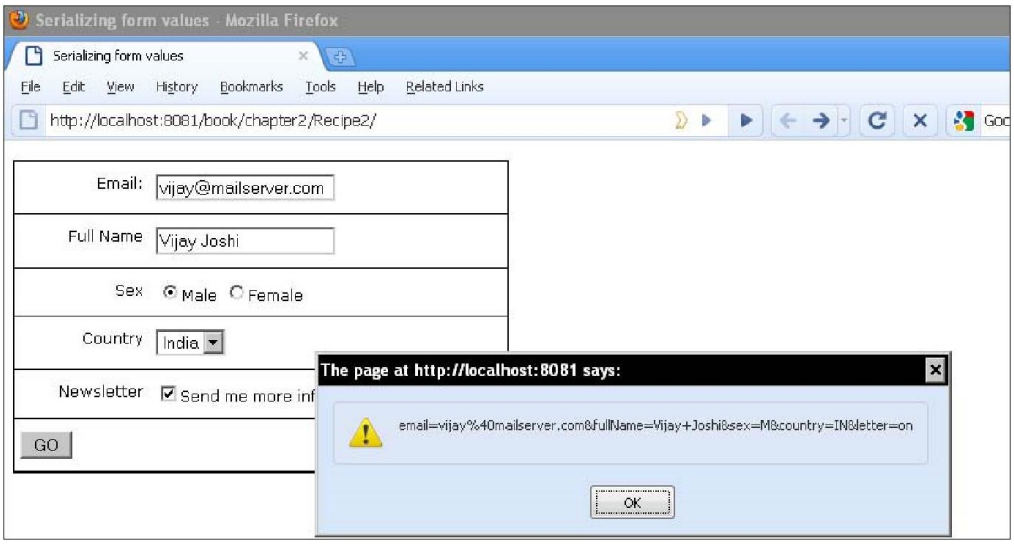
```
  <title>Serializing form values</title>
  <style type="text/css">
   ul{ border:1px solid black; list-style: none;
      margin:0pt;padding:0pt;float:left;font-family:Verdana,
      Arial, Helvetica, sans-serif;font-size:12px;width:400px;
    }
   li{  padding:10px 5px;border-bottom:1px solid black;}
   label{width:100px;text-align:right;
        margin-right:10px;float:left;}
  </style>
 </head>
 <body>
  <form>
   <ul>
    <li><label>Email:</label>
      <input type="text" name="email"/></li>
    <li><label>Full Name</label>
      <input type="text" name="fullName"/></li>
    <li>
     <label>Sex</label>
     <input type="radio" name="sex" value="M"/>Male
     <input type="radio" name="sex" value="F"/>Female
    </li>
    <li>
     <label>Country</label>
     <select name="country">
      <option value="IN">India</option>
      <option value="UK">UK</option>
      <option value="US">USA</option>
     </select>
    </li>
    <li>
     <label>Newsletter</label>
     <input type="checkbox" name="letter"/>Send me more
       information</li>
    <li>
     <input type="button" value="GO"/>
    </li>
   </ul>
  </form>
 </body>
</html>
```

2. Once again include the link to the jQuery file. After that add an event handler for the input button that we have placed on the form. This button will use the serialize() method on the form and will alert the resulting query string.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('input:button').click(function()
  {
   alert($('form:first').serialize());
```

```
  });
 });
</script>
```

3. Open your browser and run the index.html file. Fill the form and click on the GO button. The browser will display the values of form elements in a query string format, as shown in the following screenshot:



The serialize() method of jQuery turns form elements into query string format. Rather than getting each value manually and creating a query string, this function can be very handy when you want to send the values of all form elements as a part of AJAX requests. You can use any of the methods like GET or POST to send this data to the server.

serializeArray() method

Another useful function for getting values of form elements is serializeArray(). This function turns all the elements into a JavaScript object.

```
var data = $('form:first').serializeArray();
```

If the form has two textboxes named input1 and input2 and their values are value1 and value2 respectively then the object will be created as shown below:

```
[
 { input1: 'value1'  },
 { input2: 'value2'  },


]
```

Not all values are serialized

Remember that Submit buttons and File select elements are not serialized.

Name should be provided to elements

In order to successfully serialize elements do not forget to assign a name attribute to them.
If an element has been assigned an ID but not a name, it will not get serialized.

Fetching data from PHP using jQuery

Sending data to PHP

After going through this recipe you will be able to distinguish between AJAX requests and simple HTTP requests in your PHP code.

Create a new directory named Recipe3 in the chapter2 directory. Inside it create an HTML file named index.html and another PHP file check.php.

1. Open the index.html file and create a button that will load a string from a PHP file using the $.get() method.

```
<html>
 <head>
  <title>Detecting AJAX Requests</title>
 </head>
 <body>
  <form>
   <p>
    <input type="button" value="Load Some data"/>
   </p>
  </form>
 </body>
</html>
```

2. Next, include jQuery and write the code for a click event of the button. Clicking on the button will simply send an AJAX request to check.php, which will return a string. The response string will be appended to the page after the input button.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('input:button').click(function()
  {
   $.get(
    'check.php',
    function(data)
    {
     $('input:button').after(data);
    });
  });
 });
</script>
```

3. To validate that the request is indeed an AJAX request and not a direct one from the browser, open the check.php file and write the following code:

```
<?php
 if(isset($_SERVER['HTTP_X_REQUESTED_WITH']) &&  $_SERVER['HTTP_X_REQUESTED_WITH'] == 'XMLHttpRequest')
 {
   echo 'YaY!!! Request successful.';
 }
 else
 {
   echo 'This is not an AJAX request. This page cannot be accessed directly.';
 }
?>
```

4. Run the index.html file in a browser and click on the Load Some Data button.
   You will see the text YaY!!! Request successful. inserted after the button. Now
   in another window enter the direct path to the check.php file. You will see the following message:

   This is not an AJAX request. This page cannot be accessed directly.

Browsers send HTTP headers with every request that goes to a server. To distinguish between normal requests and AJAX requests, modern libraries send an additional header with AJAX request. The header's name is X-Requested-With and its value is XMLHttpRequest.

Superglobal $_SERVER contains the headers sent with the request. In the example, we have checked whether the $_SERVER array has an entry for the HTTP_X_REQUESTED_WITH key or not. If an entry is found and its value is XMLHttpRequest, we can assume it is an AJAX request. Depending upon the result of the if expression we display the resulting string
to the user.

Don't rely on X-Requested-With alone

jQuery and most of the other modern libraries (such as Prototype and Dojo) send an X-Requested-With header for the ease of the server. However, relying on this header alone is not recommended.

This is due to the reason that HTTP headers can be easily spoofed. So a user can send a request with this header that the code will assume to be an AJAX request but that won't be.

There are other ways through which you can ensure the request is legitimate but that is beyond the scope of this book.

GET and POST are the two most frequently used methods for accessing pages. In the first recipe you learned to make requests using GET method.

This recipe will make use of jQuery's $.post() method to retrieve data from a PHP script.
We will see a simple example where we will fill some data in a form and the data will be sent to PHP using the POST method. Sent data will be processed by PHP and then displayed in
the browser.

Create a new directory named Recipe4 under the chapter2 directory.

1. Create a file named index.html in the newly created Recipe4 directory. In this recipe, we will use the same form that we created in the second recipe (Creating query string automatically for all form elements) of this chapter. So write the HTML that will create a form with multiple controls.

```
<html>
 <head>
  <title>Sending data through post</title>
  <style type="text/css">
   ul{ border:1px solid black; list-style:none;
     margin:0pt;padding:0pt;float:left;
       font-family:Verdana, Arial, Helvetica,
       sans-serif;font-size:12px;width:400px;  }
   li{padding:10px 5px;border-bottom:1px solid black;}
   label{width:100px;text-align:right;margin-right:10px;
        float:left;}
   #response {display:none;}
  </style>
 </head>
 <body>
  <form>
   <ul id="information">
    <li><label>Email:</label>
      <input type="text" name="email"/></li>
    <li><label>Full Name</label>
      <input type="text" name="fullName"/></li>
    <li>
     <label>Sex</label>
     <input type="radio" name="sex" value="Male"
       checked="checked"/>Male
     <input type="radio" name="sex" value="Female"/>Female
    </li>
    <li>
     <label>Country</label>
     <select name="country">
      <option value="India">India</option>
      <option value="UK">UK</option>
      <option value="US">USA</option>
     </select>
    </li>
    <li>
     <input type="button" value="GO" name="submit"/>
    </li>
   </ul>
   <p id="response"></p>
  </form>
 </body>
</html>
```

2. Include jQuery and after that attach an event handler for the button. Clicking on the button will send an AJAX request to a PHP file using the HTTP POST method. Upon successful completion of the request, the form will be made hidden and the response received from PHP will be inserted into a paragraph.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('input:button').click(function()
  {
    var data = $('form:first').serialize();
    $.post(
     'process.php',
     data,
     function(data)
     {
      $('#information').hide();
      $('#response').html(data).show();
     },
        'html'
    );
  });
 });
</script>
```

3. Since the request is made to a PHP file, first of all create a file named process.php in the same directory as index.html. The code in this file will create a string using the data filled in the form by the user. This string will be sent back to the browser
to notify the user of the values they entered.

```
<?php
 $responseString = 'Dear '.$_POST['fullName'].', Your contact information has been saved.';
 $responseString.= 'You entered the following information: ';
 $responseString.= '<br/>';
 $responseString.= '<strong>E-mail:</strong> '.$_POST['email'];
 $responseString.= '<br/>';
 $responseString.= '<strong>Sex:</strong> '.$_POST['sex'];
 $responseString.= '<br/>';
 $responseString.= '<strong>Country:</strong> '.$_POST['country'];
  header('Content-type:text/html');
 echo $responseString;
?>'
```

4. Run the file index.html in your browser and you will see the form with some fields. Fill the value in fields and click on the GO button. You will see that the form will be hidden and the entered values will be displayed in the form as follows:

   Dear Ajay Joshi, Your contact information has been saved.You entered the following information:
   E-mail: test@test.com
   Sex: Male
   Country: India

We have registered a click event handler for GO button. Clicking the button sends a POST request to server using jQuery's $.post() method.

$.post() is almost similar to $.get() except for a couple of differences. The first, and obvious difference, is the method used which is POST for $.post() and GET for $.get(). The second difference is that POST requests are not

cached whereas GET requests are cached by the browser. Therefore, the use of the cache option with POST request will have
no effect on the request.

Other than that, both $.get() and $.post() have the same signatures.

In our example the AJAX request goes to the process.php file with the serialized data from the form. Since it is a POST request, PHP's $_POST Superglobal is populated with form data. We then extract the fields from this array and put them in a formatted string. After we have built the string we echo it back to the browser.

On receiving a successful response, we hide the form and insert the received HTML
in a paragraph.

Alternative method for $.post()

$.post(), $.get(), and other shortcut methods can also be implemented using the $.ajax() method. Given below is the $.post() implementation using $.ajax().
We will see other usage of $.ajax() in the coming recipes.

```
$.ajax(
{
  url: 'process.php',
  method: 'post',
  data: $('form:first').serialize(),
   dataType: 'html',
  success: function(response)
  {
    $('#information').hide();
    $('#response').html(response);
  }
});
```

Since $.ajax() gives more flexibility than $.post(), you can use it when you want to have a specific error callback function for request.

Fetching data from PHP using jQuery explains the $.get() method in detail

Creating a query string automatically for all form elements

Handling errors in AJAX requests, which shows how to handle errors encountered during AJAX requests

Consider a case where a user is allowed to select a date on a page and an AJAX request is made to the server to fetch some data against that date. If the request is under processing and in the meantime the user selects another date and a new request is sent, the server now has two requests pending.

Imagine what will happen to an application if there are multiple users repeating the same behavior. Desirable behavior in this case will be to cancel the pending request and allow
only the current one.

This recipe will explain how to cancel any pending requests.

Create a new folder in chapter2 directory and name it as Recipe5.

1. We will use the same markup that we created in the first recipe of this chapter. So create a new file index.html and write the code to create an HTML page with a combo box and two options. Also create a paragraph element on the page that will display the received response.

```
<html>
 <head>
  <title>Aborting ajax requests</title>
  <style type="text/css">
   ul{border:1px solid black; list-style:none;
      margin:0pt;padding:0pt;float:left;
      font-family:Verdana, Arial, Helvetica, sans-serif;
      font-size:12px;width:300px;}
   li{padding:10px 5px;border-bottom:1px solid black;}
  </style>

 </head>
 <body>
  <form>
   <p>
    Show list of:
    <select id="choice">
     <option value="">select</option>
     <option value="good">Good Guys</option>
     <option value="bad">Bad Guys</option>
    </select>
   </p>
   <p id="response"></p>
  </form>
 </body>
</html>
```

2. Now comes the jQuery code. Define a global variable and after that attach an event handler for the combo box. The handler function checks if an AJAX request to the server is already pending or not. On finding a pending request it will abort that request and a new request will be sent to the server.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  var ajax;
  $('#choice').change(function()
  {
   if(ajax)
   {
    ajax.abort();
   }
   ajax = $.get(
    'wait.php',
    { what : $(this).val() },
    function(response)
    {
```

```
        $('#response').html(response);
      },
      'html'
    );
  });
});
</script>
```

3. Finally comes the PHP part. Create a PHP file and name it as wait.php. Write the same code from the recipe *Fetching data from PHP using jQuery*. The code will check for the values received from the browser and will send a response accordingly. For this example we will make PHP wait for 10 seconds before any response is sent to
the browser so that we are able to send multiple requests within 10 seconds.

```
<?php
 sleep(10);
 if($_GET['what'] == 'good')
 {
  $names = array('Sherlock Holmes', 'John Watson', 'Hercule Poirot', 'Jane Marple');
  echo getHTML($names);
 }
 else if($_GET['what'] == 'bad')
 {
  $names = array('Professor Moriarty', 'Sebastian Moran',
          'Charles Milverton', 'Von Bork', 'Count Sylvius');
  echo getHTML($names);
 }
 function getHTML($names)
 {
  $strResult = '<ul>';
  for($i=0; $i<count($names); $i++)
  {
   $strResult.= '<li>'.$names[$i].'</li>';
  }
  $strResult.= '</ul>';

  return $strResult;
 }
?>
```

4. Now run your browser and select a value from the combo box. PHP will send the response after 10 seconds. Now select another value from the combo box. The pending request will be aborted and the current request will be sent to the server. The response received will be according to the currently selected value. No response
will be received for previous selection as the request was aborted.

All AJAX methods of jQuery return an XMLHttpRequest object when called. We have declared a global variable ajax that will store this object. When a value is selected from the combo box, the handler function checks if the variable ajax is defined or not. In case of the first selection it will be undefined, hence nothing happens and the request is sent to the wait.php file. The XMLHttpRequest object created for sending this request is stored in variable ajax.

Now when a value of combo box is changed ajax will be holding the XMLHttpRequest object that was used to send the previous request. XMLHttpRequest has an abort() method that cancels the current request. In our case the

pending request to the server is cancelled and a new request is made, which is again stored in the ajax variable.

Now onwards, changing a value of combo box within 10 seconds will cancel out a pending request and will send a fresh one to the server.

Handling errors in AJAX requests

The larger a web page the more time a browser will take to download it. This may degrade the user experience in case of slow connections or larger pages.

One approach that can be followed is to load only what is absolutely necessary for the user and load the rest of the content when required. There are some sections on a page which
are rarely accessed. It will make page loads faster and user experience will improve.

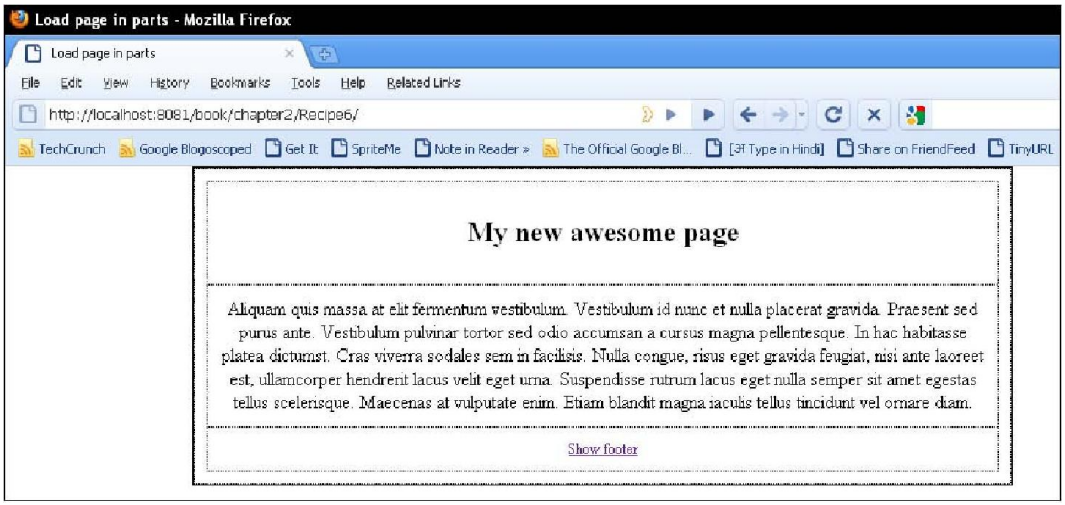
In this recipe we will demonstrate this case with a simple example. We will create a single HTML page and will allow the user to load its one section when required.

Create a folder named Recipe6 in chapter2 directory.

1. Create a new file and save it as index.html. This page will have three sections: head, content, and footer. HTML for the footer will not be created; instead we will load it dynamically. We have also applied some CSS in the head section to customize the appearance of the page.

```
<html>
 <head>
  <title>Load page in parts</title>
  <style type="text/css">
   body { border:1px solid black;margin:0 auto;text-align:center;width:700px; }
   div { padding:10px;border:1px dotted black; }
   #footer > a { font-size:12px;margin:50px; }
  </style>
 </head>
 <body>
  <div>
   <div id="head"><h2>My new awesome page</h2></div>
   <div id="content">
    <span>
    Aliquam quis massa at elit fermentum vestibulum.
    Vestibulum id nunc et nulla placerat gravida. Praesent
    sed purus ante. Vestibulum pulvinar tortor sed odio
    accumsan a cursus magna pellentesque. In hac habitasse
    platea dictumst. Cras viverra sodales sem in facilisis.
    Nulla congue, risus eget gravida feugiat, nisi ante
    laoreet est, ullamcorper hendrerit lacus velit eget urna.
    Suspendisse rutrum lacus eget nulla semper sit amet
    egestas tellus scelerisque. Maecenas at vulputate enim.
```

```
    Etiam blandit magna iaculis tellus tincidunt vel ornare
    diam.
    </span>
   </div>
   <div id="footer">
    <a href="#" id="loadFooter">Show footer</a>
   </div>
  </div>
 </body>
</html>
```



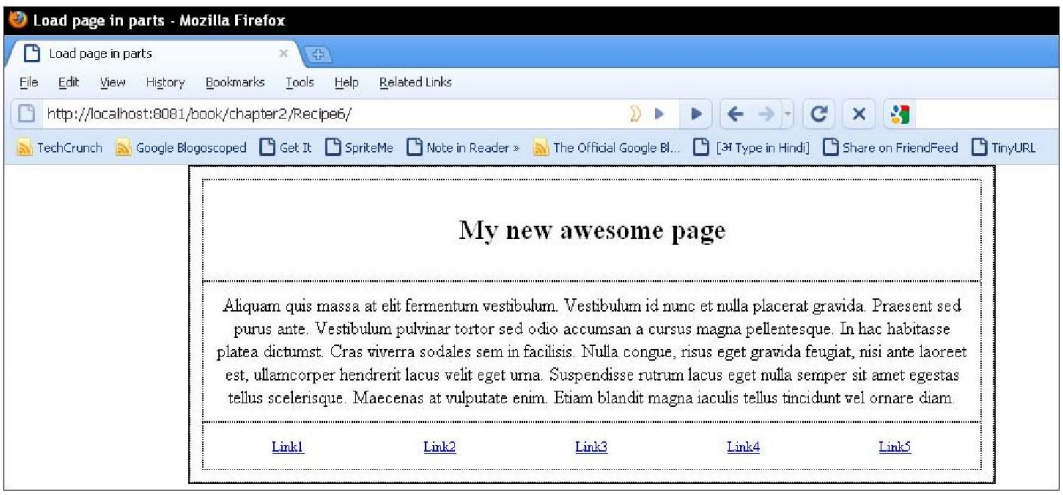
2. Next, we will need to create a file where we will write HTML for the footer. Open a new file and save it with the following markup as footer.html.

```
<a href="#">Link1</a>
<a href="#">Link2</a>
<a href="#">Link3</a>
<a href="#">Link4</a>
<a href="#">Link5</a>
```

3. To glue all the above things, switch back to index.html and write the jQuery code for the Show footer link.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#loadFooter').click(function()
  {
   $('#footer').load('footer.html');
  });
 });
</script>
```

4. Open your browser and run the index.html file. Click on the Show footer link. jQuery will load the HTML for the footer from the footer.html file and will
insert it inside the footer section.



jQuery provides a method load() that acts on HTML elements. It gets the data from the server and inserts it into the HTML element or elements that called it. load() takes three parameters. The first parameter is the URL, from where data will be loaded, the second parameter is the data that can be sent to the server. The third parameter is a callback function which executes once data has loaded.

In the previous example, clicking the Show footer link calls the load() method on element with ID footer. It loads the footer.html file in which we wrote the markup for the footer. After the file has loaded successfully its HTML is inserted into the footer.

Difference between load and get

Both these methods are similar except for the fact that load is a method, which means it acts on a set of elements specified by a selector. Once the request is complete, the HTML of elements specified by the selectors is set. On the other hand $.get is a global method that has an explicitly defined callback function.

Fetching data from PHP using jQuery
Sending data to PHP earlier in this chapter
Loading JavaScript on demand to reduce page load time, in this chapter

Errors are inevitable. Period. Sometimes things are not in your control—like server failures—and in this case you must have an error handling mechanism in place, which can catch the errors and show them to the users. Throughout the recipes in this chapter we have implemented callback functions that execute when a request is successful. It may happen (and I promise you it will happen) that you typed a filename incorrectly or the server encounters an error and you get an error rather than a successful response.

This recipe will explain how to deal with such situations in AJAX requests.

Create a folder Recipe7 inside the chapter2 folder.

1. Create a file named index.html in the Recipe7 folder. Define some CSS styles in it and create an input box that will ask for a filename to load and a button. Also create a paragraph where contents loaded in a file will be displayed.

```
<html>
 <head>
  <title>Error handling</title>
  <style type="text/css">
   ul{ border:1px solid black; list-style:none;margin:0pt;
      padding:0pt;float:left;font-family:Verdana,
      Arial, Helvetica, sans-serif;font-size:12px;width:300px;
    }
   li{  padding:10px 5px;border-bottom:1px solid black;}
   span{ color:red;}
  </style>
 </head>
 <body>
  <label for="fileName">Enter file name to load: </label>
  <input type="text" id="fileName"/>
  <input type="button" value="Load file"/>
  <p id="result"></p>
 </body>
</html>
```

2. Before the body tag closes, include jQuery and write code using the $.ajax() method that will fire an AJAX request to load the file specified by the user.
   Define both success and error callbacks here.

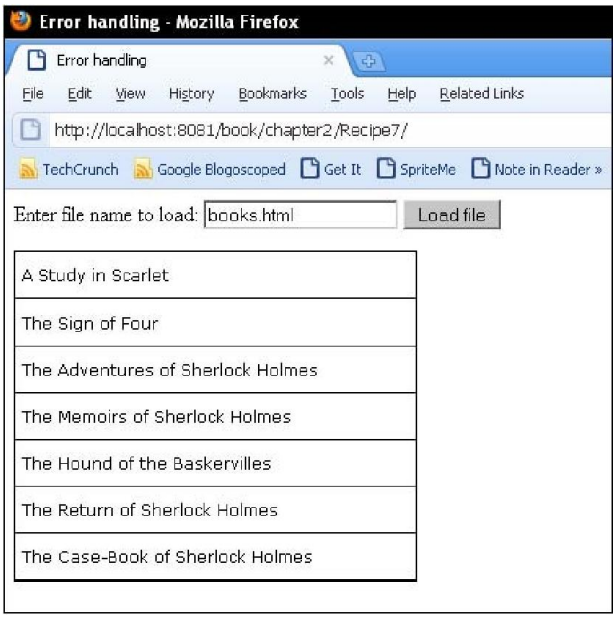
```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
$(document).ready(function ()
{
 $('input:button').click(function()
 {
  if($('#fileName').val() == '')
  {
   $('#result').html('<span>Please provide a file
     name.</span>');
   return;
  }
  $.ajax({
  url: $('#fileName').val(),
  method: 'get',
  success: function(data)
   {
    $('#result').html(data);
   },
  error : function()
   {
    $('#result').html('<span>An error occured.</span>');
   }
```

```
  });
 });
});
</script>
```

3. Create another HTML file and name it as books.html. In this file create an unordered list of books, as follows:

```
<ul>
 <li>
  A Study in Scarlet
 </li>
 <li>
  The Sign of Four
 </li>
 <li>
  The Adventures of Sherlock Holmes
 </li>
 <li>
  The Memoirs of Sherlock Holmes
 </li>
 <li>
  The Hound of the Baskervilles
 </li>
 <li>
  The Return of Sherlock Holmes
 </li>
 <li>
  The Case-Book of Sherlock Holmes
 </li>
</ul>
```

4. Launch your browser and run the index.html file. Enter books.html in the textbox and click on the Load file button. jQuery will send an AJAX request and you will see a nicely formatted list of books on your screen. Leaving the field blank and clicking on the Load File button will display an error.



5. Now enter the name of any non-existent file such as none.html or nofile.html. Clicking on the Load file button will display an error.

In this example we used the low level AJAX implementation of jQuery. Other methods like $.get(), $.post(), and so on are task-specific implementations of $.ajax(). As you just saw $.get() is specific to GET requests whereas another method $.getScript() is used only for loading scripts.

One of the many options of $.ajax() is the error callback. When a request fails due to some reason like a missing file, timeout on server, or a server error this callback executes, whereas higher-level implementations do not take any action in this case.

In the previous example, we have used the error callback to display an error message to the user. We intentionally typed a filename that does not exist and jQuery passed the control
to the error callback.

Parameters passed to error callback

jQuery makes three parameters available to the error callback. These are the XMLHttpRequest object that was used to send a request, a string indicating the type of error, and an exception
(if any) from the JavaScript side.

The second parameter is a string that can be one of these: timeout, error, notmodified, parsererror, or null.

The ajaxError() method

Another method ajaxError() is available that can be attached to HTML elements. This method will execute every time there is an error in AJAX request.

```
$('#result').ajaxError(function()

{

  $(this).html('<span>An error occured.</span>');
})
```

Place this code inside document.ready() and then remove the error callback from the function's definition. Now enter an incorrect filename and click on the button. You will still see

an error.

This method can be pretty useful when you have AJAX requests originating from multiple places in a page and you want a single placeholder for error messages. The error message will be displayed each time because it will be executed regardless of where the request originated.

Fetching data from PHP using jQuery
Sending data to PHP
Creating an empty page and loading it in parts
Loading JavaScript on demand to reduce page load time

In case of GET requests, browsers cache these requests and when the request is invoked again they do not send the request to the server and instead serve it from the cache.

This recipe will explain how to force browsers to send the request to a server instead of serving it from the cache.

1. While sending an AJAX request use the cache option to force no caching by the browser. Setting the cache option to false does not let the browser cache any AJAX requests and the data is loaded from the server each time the request is made.

```
$.ajax({
  url : 'someurl.php',
  cache: false,
  success: function(data)
  {
    //do something with received data
  }
});
```

On an AJAX request, the browser checks if a request to that URL is already in the browser cache or not. If it is found in the cache, no request to the server is sent and response from
the cache is served.

jQuery provides a cache option that can be used to override this browser behavior. By default, cache is set to true. When this option is set to false, jQuery appends an underscore key
(_) with a random numeric value to the URL. This makes the browser assume that each URL is unique even when only the value of the underscore key is different. Hence, the browser does not cache the request and it goes to the server each time.

Only GET requests are cached

It is worth noting that only GET requests are cached by the browser and not POST requests. Therefore, using the cache option with POST requests will have no effect. Every POST request
is a fresh request.

Fetching data from PHP using jQuery explains $.get() method for making
get requests

Sending data to PHP explains the $.post() method for making POST requests

Think of a rich Internet application that makes heavy use of JavaScript to interact with the user. Such a page typically consists of more than one JavaScript files, such as a file for calendar control, another file for special effects, yet another plugin for your cool accordion, and so on.

This results in the increase of the page load time as browsers cannot download all of these files simultaneously. The best solution for this is to load only absolutely necessary files at the
time of loading the page and load the other files when required.

This recipe will explain how JavaScript files can be loaded on demand.

Create a directory named Recipe9 in the chapter2 folder.

1. Create a file index.html in the chapter2 folder. Write the HTML to create a page that will have a paragraph element and four buttons. The first button will be used to load another JavaScript file and rest of the buttons will manipulate the paragraph.

```
<html>
 <head>
  <title>getScript example</title>
 </head>
 <body>
  <p id="container">
   This text will be replaced with new text.
  </p>
  <input type="button" class="loader" value="Load Script"/>
  <input type="button" class="bold" value="Bold"/>
  <input type="button" class="color" value="Change color"/>
  <input type="button" class="change" value="Change text"/>
 </script>
 </body>
</html>
```

2. Before the body tag closes, include the jQuery library and add event handler for the first button. On click of the button, jQuery will load a JavaScript file. On successful loading of the JavaScript, a function named addEvents() will be called that will
add event handlers for all other buttons.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
$(document).ready(function ()
{
 $('input:button:first').click(function(aaa)
 {
  $.getScript('new.js', function()
  {
   alert('Script loaded');
   addEvents();
  });
 });
});
</script>
```

3. Now create a new file in the same directory as index.html and name it as new.js. Define the function addEvents() in it to add events for the four buttons.

```
function addEvents()
{
 $('.bold').click(function()
  {
   $('#container').css('font-weight', 'bold');
  });

 $('.color').click(function()
  {
   $('#container').css('color', 'red');
  });

 $('.change').click(function()
  {
   $('#container').html('<em>New html inserted</em>');
  });
}
```

4. Open your browser and run the index.html file. Click on any of the buttons except Load Script. You will find that nothing happens to the paragraph's content. Now click on the Load Script button. An alert will appear notifying that script has been loaded. Clicking on any of the last three buttons will now change the appearance of
the paragraph.

Clicking on the Load Script button invokes the $.getScript() method of jQuery. This function has two parameters: the file name to be loaded and a callback function that
executes when the file is successfully loaded.

It loads the specified JavaScript file asynchronously from the server. After a successful load, all the variables and functions of that file are available in the global context. This means they can be used by other JavaScript files too. A successful callback ensures that the file has been loaded and, therefore, we can safely work with the variables or functions of that file.

In the previous example the function addEvents() is defined in the new.js file. This function binds event handlers to our buttons. Since new.js is not available on the page, these buttons do nothing. After the file is loaded, we call the addEvents() function, which binds these buttons to respective events. Thus, these buttons become functional.

Alternative method for getScript

The $.getScript() method is specifically for loading scripts only. It can be written using the $.ajax() method too.

```
$.ajax(
{
  url: 'new.js',
  dataType: 'script',
  success: function()
  {
    alert('Script loaded');
    addEvents();
  }
});
```

The above code will also load the new.js file and execute it. Use this method if you need the error callback too, which is not available with $.getScript().

Also note the use of the dataType option here. We have provided its value as script. The dataType parameter tells jQuery what type of data to expect from the server (which is script in this case).

Fetching data from PHP using jQuery explains get method for fetching data

Sending data to PHP explains how to send data to PHP through jQuery

Creating an empty page and load it in parts

5

Working with Forms

In this chapter, we will cover:

Adding input fields dynamically in a form
Searching for a user-inputted string in a page
Checking for empty fields using jQuery
Validating numbers using jQuery
Validating e-mail and website addresses using regular expressions
Displaying errors as user types: performing live validation
Strengthening validation: validating again in PHP
Creating a voting system
Allowing HTML inside textareas and limiting HTML tags that can be used

Forms and pages are the only part of your web application that the end-user uses directly. It is, therefore, the responsibility of a web developer to make forms that are easy to use, easy to navigate, and interactive. Moreover, attackers can try to damage your application by trying to input malicious data through your forms.

This chapter deals with forms and form validations like searching for data in a form both on the browser and the server side. Though validation can be done on the browser with the help of jQuery, validating data on the server side is more important. If JavaScript is disabled on the browser, then the client-side validation will not work. Validation on the client side makes your application user-friendly and less error prone. You will learn how to validate forms for different types of data such as empty fields, numbers, e-mail or web addresses, and so on later in this chapter.

Validation on the server side is a must and the client-side validation should not be seen as a replacement for it because client-side validation can be disabled.

We will create a form where you will be able to add more fields to a form without making a trip to the server side. In our form, we will present the user with a single textbox and we will provide buttons for adding and removing additional textboxes.

Create a folder for this recipe named Recipe1 in the Chapter5 directory. Do not forget to put the jquery.js file inside the Recipe1 folder.

1. Create a file and save it as index.html in the Recipe1 folder and write an HTML code that will create a list with only one list item. This list item will only have a single textbox. In the end, we'll see a button that will add more fields to our form.

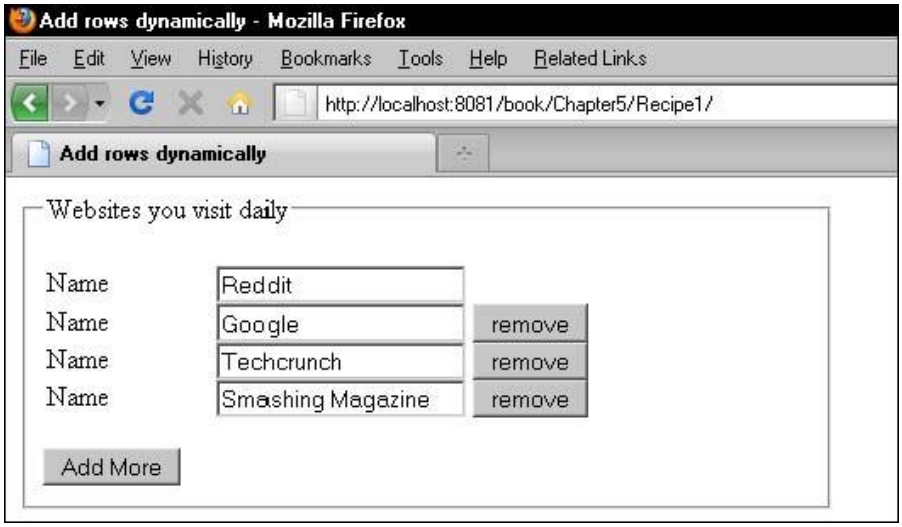
```
<html>
 <head>
  <title>Add rows dynamically</title>
  <style type="text/css">
   fieldset{width:450px;}
   ul{padding:2px;list-style:none;}
   label{float:left;width:100px;}
  </style>
 </head>
 <body>
  <form action="process.php" method="post">
   <fieldset>
    <legend>Websites you visit daily</legend>
    <ul id="sites">
     <li>
      <label>Name</label><input type="text" value=""/>
     </li>
    </ul>
    <input type="button" id="add" value="Add More"/>
   </fieldset>
  </form>
 </body>
</html>
```

2. Now, include jQuery and write event handlers. The first event handler will be for the Add More button that will add more textboxes and also a button to remove them. We will write another event handler that will remove selected textboxes.

```
<script type="text/javascript" src="../jquery.js"></script>
```

```
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#add').click(function()
  {
   var str = '<li>';
   str+= '<label>Name</label><input type="text" value=""/> ';
   str+= '<input type="button" value="remove"
         class="remove"/>';
   str+= '</li>';
   $('#sites').append(str);
  });

  $('.remove').live('click', function()
  {
   $(this).parent('li').remove();
  });
 });
</script>
```

3. Run the file in your browser. Clicking on the Add More button will add more textboxes to the page. You can also remove specific textboxes by clicking on the remove button next to the textbox.



The event handler for the Add More button creates a new li with a textbox and a remove button inside it, and then uses jQuery's append method to append it to an existing list
of sites.

Note that all remove buttons have a class called remove specified. We used the live method to attach the event handler for elements having this class. If you remember, the
live method adds an event handler to those elements that are already present in the form as well as those that will be created in the future.

Therefore, clicking on the remove button on any row finds its parent li and removes it from the DOM.

Getting values on server side

All these textboxes are generated on the client side that is using jQuery. To access these on the server side, all of these should have a name attribute. Since all of these belong to the same group (websites), we can provide a name attribute in array format that will allow us
to get all filled values in the form of an array.

Simply add name="sites[]" to the existing textbox as well as when we create it from jQuery. Now if the form is submitted you can access all the filled values from the array $_POST['sites']. Given below is the $_POST array after submitting the form with
some values:

```
Array
(
   [sites] => Array
      (
         [0] => Purple
         [1] => Violet
         [2] => Red
         [3] => Green
         [4] => Yellow
      )



)
```

We will use jQuery to highlight a word entered by the user. The data on the browser can be made available from the server side (or database) as well. For this example, we will use some
text in an HTML page. The user will enter a search query in a textbox and after pressing a button all matching words in the content will be highlighted.

Create a folder for this recipe in the Chapter5 directory and name it as Recipe2.

1. Open a new file, name it as index.html and save it in the Recipe2 folder. Let us begin by writing the markup now. Create some paragraphs and put some text inside them. In the end, place a textbox and two buttons. We have also defined a CSS class highlight that will create the highlight effect.

```
<html>
 <head>
  <title>Search</title>
  <style type="text/css">
   p { border:1px solid black;width:500px;padding:5px;}
   .highlight { background-color:yellow; }
  </style>
 </head>
 <body>
```

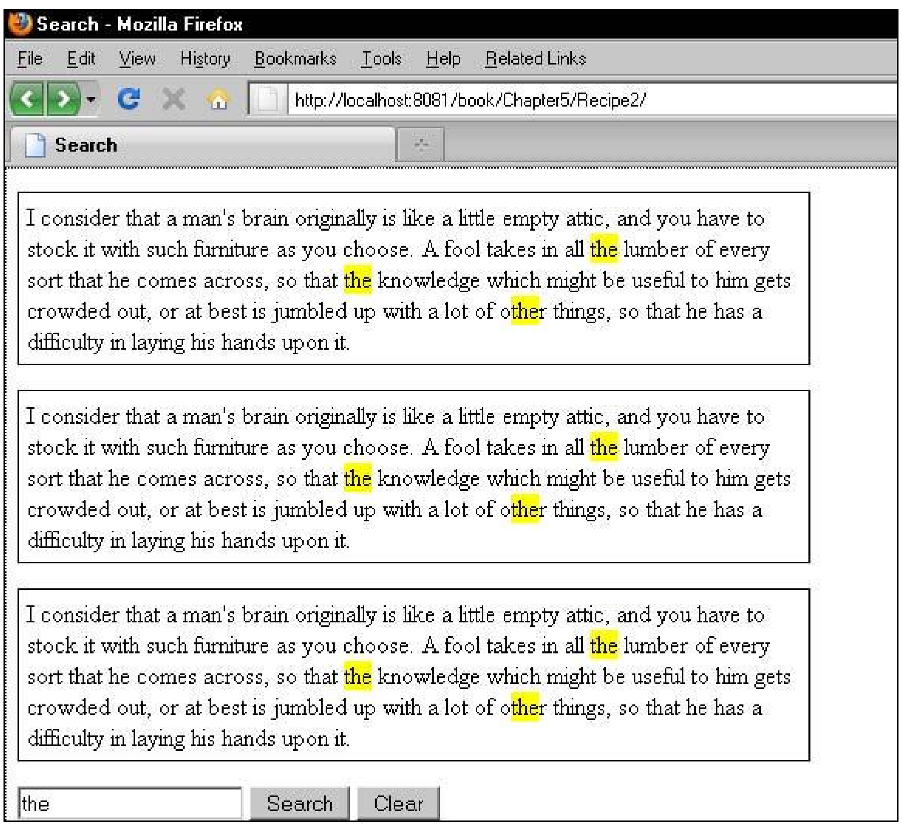
```
    <form>
      <p>
        I consider that a man's brain originally is like a little
        empty attic, and you have to stock it with such furniture
        as you choose. A fool takes in all the lumber of every
        sort that he comes across, so that the knowledge which
        might be useful to him gets crowded out, or at best is
        jumbled up with a lot of other things, so that he has a
        difficulty in laying his hands upon it.
      </p>
      <p>
        I consider that a man's brain originally is like a little
        empty attic, and you have to stock it with such furniture
        as you choose. A fool takes in all the lumber of every
        sort that he comes across, so that the knowledge which
        might be useful to him gets crowded out, or at best is
        jumbled up with a lot of other things, so that he has a
        difficulty in laying his hands upon it.
      </p>
      <p>
        I consider that a man's brain originally is like a little
        empty attic, and you have to stock it with such furniture
        as you choose. A fool takes in all the lumber of every
        sort that he comes across, so that the knowledge which
        might be useful to him gets crowded out, or at best is
        jumbled up with a lot of other things, so that he has a
        difficulty in laying his hands upon it.
      </p>
      <input type="text" id="text"/>
      <input type="button" id="search" value="Search"/>
      <input type="button" id="clear" value="Clear"/>
    </form>
  </body>
</html>
```

2. Before the body tag closes, include jQuery. Now in the form we have two buttons. The first button is for searching the entered text and the second one is for clearing the highlighted parts. For searching, we'll call a highlight function by clicking on the Search button. This function searches the text on the page and on finding it, wraps it into HTML tags and applies the highlight class to it. The second button calls the clearSelection function that restores the page to normal.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
  $(document).ready(function ()
  {
    $('#search').click(highlight);
    $('#clear').click(clearSelection);

    function highlight()
    {
      var searchText = $('#text').val();
      var regExp = new RegExp(searchText, 'g');
      clearSelection();
      $('p').each(function()
      {
        var html = $(this).html();
```

```
    var newHtml = html.replace(regExp,
        '<span class="highlight">'+searchText+'</span>');

    $(this).html(newHtml);
   });
 }

  function clearSelection()
  {
   $('p').each(function()
   {
    $(this).find('.highlight').each(function()
    {
     $(this).replaceWith($(this).html());
    });
   });
  }
 });
</script>
```

3. Run the file in your browser and enter a search term in the textbox. Click on the Search button and all matching words will be highlighted on the page. Click on the Clear button to reset.



After entering a search term and clicking on the Search button, the highlight function is called. This function first

clears any highlights on the page by calling the clearSelection function. We will see what clearSelection does in a moment. Next, we get the entered search term in variable searchText. After that, we create an object using the RegExp
method of JavaScript. This regular expression will perform an actual search for the entered text.

Then we iterate through each paragraph on the form. We get the HTML of each paragraph and we get to use JavaScript's replace function on that HTML. The replace function takes two parameters. The first parameter is the regular expression object and the second one is the text with which we have to replace the matched text. We have just wrapped the search text in a span and assigned CSS class highlight to it. The replace function will return the whole text with the replaced words. We then replace the original HTML of the current paragraph with this new one.

Search and replace

You can extend this idea and could create a simple utility for "search and replace". Rather than highlighting the selected text, you can ask for a string to replace it with.

Validation is an important technique in client-side scripting. Validation on the client side can significantly reduce round trips to the server by providing instant feedback in the form of messages. Even so, it is NOT recommended to rely on the client-side validation alone. JavaScript on the users' browsers might be turned off; therefore, validation should ALWAYS
be done again on the server side as well.

1. Create a file for this recipe and name it index.html. Create a form with some text fields and an input button. Note that all textboxes except city has a class name required assigned to them. This will be used while validating the fields.

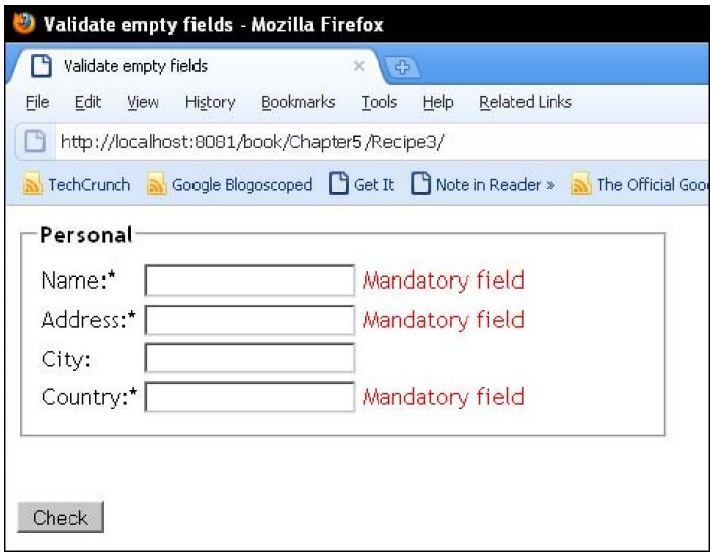
```
<html>
 <head>
  <title>Validate empty fields</title>
  <style type="text/css">
   body{font-family:"Trebuchet MS",verdana;width:450px;}
   .error{ color:red; }
   #info{color:#008000;font-weight:bold; }
  </style>
 </head>
 <body>
  <form>
   <fieldset>
    <legend><strong>Personal</strong></legend>
    <table>
     <tbody>
      <tr>
      <td>Name:* </td>
      <td><input type="text" class="required" /></td>
      </tr>
      <tr>
      <td>Address:* </td>
      <td><input type="text" class="required"/></td>
```

```
        </tr>
        <tr>
          <td>City: </td>
          <td><input type="text"/></td>
        </tr>
        <tr>
          <td>Country:* </td>
          <td><input type="text" class="required"/></td>
        </tr>
       </tbody>
     </table>
    </fieldset>
    <br/>
    <span id="info"></span>
    <br/>
    <input type="button" value="Check" id="check" />
   </form>
  </body>
</html>
```

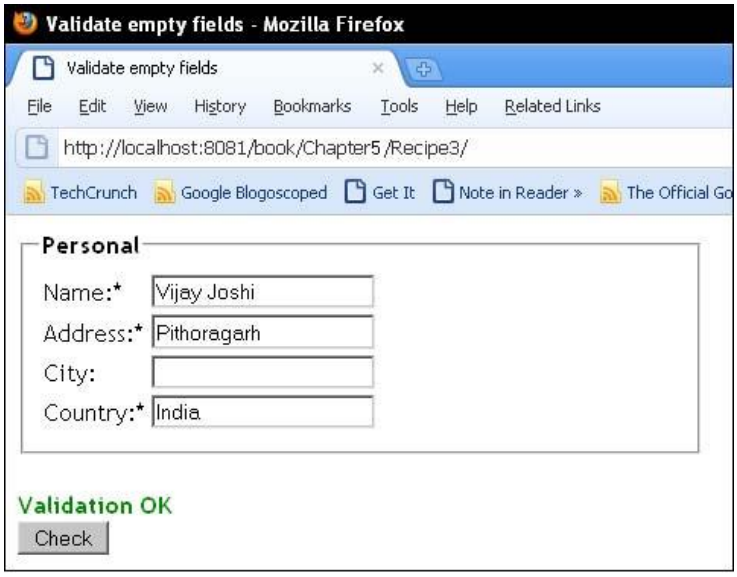
2. Now, include the jQuery before the <body> tag closes. Write the validation code that attaches a click event handler to the input button.The validate function will be called on clicking this button that will check the text fields for empty values.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
   $('#check').click(validate);
   function validate()
   {
     var dataValid = true;
     $('#info').html('');
     $('.required').each(function()
     {
       var cur = $(this);
       cur.next('span').remove();
       if ($.trim(cur.val()) == '')
       {
         cur.after('<span class="error"> Mandatory field
                 </span>');
         dataValid = false;
       }
     });
     if(dataValid)
     {
       $('#info').html('Validation OK');
     }
   }
 });
</script>
```

3. Launch your browser and run the index.html file. Try clicking on the Check button without filling in values for the textboxes. You will see an error message next to each textbox that needs to be filled:



After filling the required values in each of the textboxes, click on the button again and this time you will see the Validation OK message appearing above the Check button as shown
in the following screenshot:



We start by assigning a class name required to each textbox that we wish to make mandatory. This way we will be able to use jQuery's class selector to select all such textboxes.

First of all, in the jQuery code, we have attached an event handler to the Check button that calls the validate function. This function starts by declaring a variable dataValid to true and then it selects all the textboxes that have CSS class required. It then iterates in this collection and removes any span elements next to the textbox. These span elements maybe previous error messages. If we do not remove them, we will have multiple similar looking error messages next to a single textbox.

After this, the if condition checks the value of the current textbox. Note the use of jQuery utility function trim here. Since blank spaces are not considered valid values, we trim these from the text value. If a blank value is found, we

append a span with an error message next
to the current textbox and variable dataValid is set to false.

After all the iterations are done using jQuery's each method, we check the value of dataValid. If it's still true, that means no field is blank and we display a Validation OK message on the screen.

Validating fields one by one

If you do not want to show all errors at once but instead want to make sure that the user has filled the first field and then proceeded to the next, you can do so by modifying the previous code.

To do that, change the if condition as follows:

```
if ($.trim(cur.val()) == '')
{
  cur.after('<span class="error"> Mandatory field</span>');
  dataValid = false;
}
```

And remove this code:

```
if(dataValid)
{
  $('#info').html('Validation OK');


}
```

Validating numbers using jQuery

Validating e-mail and website addresses using regular expressions

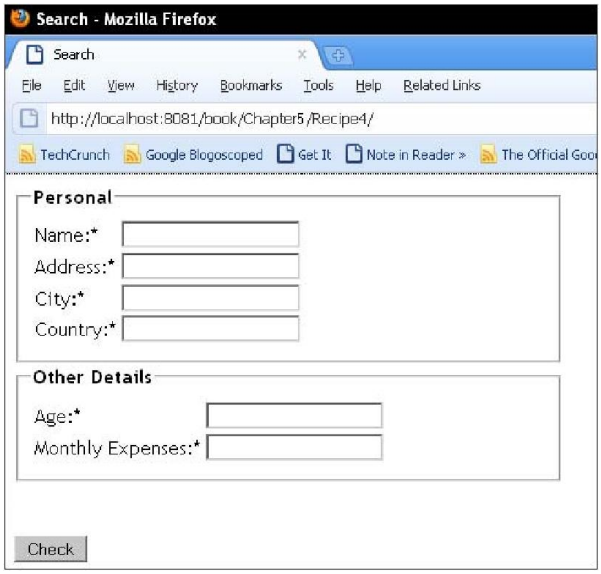
Displaying errors as user types: *performing live validation*

In the last recipe, we validated empty fields. In this recipe, we will extend that behavior and will check for numbers along with empty fields.

Create a new folder Recipe4 inside the Chapter5 directory.

1. Create a new file and save it as index.html in the Recipe4 folder. We will take the same code form as used in the previous recipe and will add another section to it. So, copy the code from the previous recipe to the index.html file. Now, we will add another section to it through which a user will be able to enter some numbers. Create another section named Other Details after the Personal section. It is important to note that these fields have another CSS class named number along with required assigned to them. This way we will be able to validate for empty fields as well as
   for numbers.

```
<fieldset>
 <legend><strong>Other Details</strong></legend>
 <table>
  <tbody>
   <tr>
   <tr>
    <td>Age:* </td>
    <td><input type="text" class="required number"/></td>
   </tr>
   <tr>
    <td>Monthly Expenses:* </td>
    <td><input type="text" class="required number"/></td>
   </tr>
   </tr>
  </tbody>
 </table>
</fieldset>
```



2. Now, let's look at the jQuery code. Once again, include the jQuery library and write the code for validating empty fields as well as numbers. Clicking on the button this time will first check for blank fields. If any of the fields are empty, the user will be notified and we will jump out of the function. Once all the fields have passed the blank field validation, jQuery will check for those textboxes that should have numbers only. Here is the complete jQuery code:

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#check').click(validate);

  function validate()
  {
   var dataValid = true;
   $('.required').each(function()
   {
    var cur = $(this);
```

```
      cur.next().remove();
      if ($.trim(cur.val()) == '')
      {
        cur.after('<span class="error"> Mandatory field
              </span>');
        dataValid = false;
      }
    });
    if(!dataValid) return false;

    $('.number').each(function()
    {
      var cur = $(this);
      cur.next().remove();
      if (isNaN(cur.val()))
      {
        cur.after('<span class="error"> Must be a number
              </span>');
        dataValid = false;
      }
    });
    if(dataValid)
    {
      $('#info').html('Validation OK');
    }
   }
  });
</script>
```

In the previous code, we first check for empty fields by iterating on elements with class name required. After the iterations are complete we check the value of the dataValid field. If it is false, we'll return immediately from the function. Once all the fields are non-empty, we proceed to check for numbers.

We select all the elements with class name or number and use the each method to check each element. JavaScript function isNaN (is Not a Number) can be used to determine if a value is a number or not. If a value is found that is not a number, we append the appropriate error message after that element.

If all elements pass this validation, the message Validation OK gets displayed near the Check button.

Checking for empty fields using jQuery

Validating e-mail and website addresses using regular expressions

Displaying errors as user types: performing live validation

While filling out web forms it is common to ask a user for an e-mail ID and a website name. These values are a little bit different from the normal strings as they have a fixed pattern. E-mail addresses require @ symbol whereas website addresses generally start with http or https. These and many other conditions are required by such

addresses.

This is where regular expressions come to the rescue. This recipe will show you the use of regular expressions to validate patterns like e-mail addresses and URLs.

Create a new folder named Recipe5 inside the Chapter5 directory.

1. Create a file named index.html inside the Recipe5 folder. Similar to the previous recipe, create two textboxes—one for entering the e-mail address and another for the website address. Also, assign a CSS class mail to the first textbox and site to the second one.

```
<html>
 <head>
  <title>Search</title>
  <style type="text/css">
   body{font-family:"Trebuchet MS",verdana;width:450px;}
   .error{ color:red; }
   #info{color:#008000;font-weight:bold; }
  </style>
 </head>
 <body>
  <form action="process.php" method="post">
   <fieldset>
   <legend><strong>Contact Details</strong>- both fields are
        mandatory</legend>
    <table>
     <tbody>
      <tr>
      <tr>
       <td>Email: </td>
       <td><input type="text" class="required mail"/></td>
      </tr>
      <tr>
       <td>Website:<br/>(start with http:// ) </td>
       <td><input type="text" class="required site"/></td>
      </tr>
      </tr>
     </tbody>
    </table>
   </fieldset>
   <br/>
   <span id="info"></span>
   <br/>
   <input type="button" value="Check" id="check" />
  </form>
 </body>
</html>
```

2. To make our validations actually work, first include the jQuery library. Then add an event handler for the Check button. It will first search for all elements with class name mail and will validate the entered e-mail address against a regular expression. After that, it will validate the website address entered by the user, again against a regular expression. If no match is found, an error will be displayed next to that textbox.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#check').click(validate);

  function validate()
  {
   var dataValid = true;

   $('.mail').each(function() {
    var cur = $(this);
    cur.next('span').remove();
    var emailPattern = /^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-
                z\.]{2,6})$/;
    if (!emailPattern.test(cur.val()))
    {
     cur.after('<span class="error"> Invalid Email Id
            </span>');
     dataValid = false;
    }
   });
   if(!dataValid)     return;

   $('.site').each(function() {
    var cur = $(this);
    cur.next('span').remove();
    var urlPattern = /^(http(s?))\:\/\/www.([0-9a-zA-Z\-
                ]+\.)+[a-zA-Z]{2,6}(\:[0-9]+)?(\/\S*)?$/;
    if (!urlPattern.test(cur.val()))
    {
     cur.after('<span class="error"> Invalid URL</span>');
```

```
        dataValid = false;
      }
    });
    if(dataValid)
    {
      $('#info').html('Validation OK');
    }
   }
 });
</script>
```

On clicking the Check button, the validate function is called. This function first defines the variable dataValid to true. Then it gets all textboxes with class name mail and iterates in the selection. We declare a variable emailPattern, which defines a regular expression. Then, inside the if condition, we use JavaScript test function to check the value of textbox against the regular expression. If the pattern does not match, we append an error message next to the textbox and set the dataValid variable to false.

We then repeat the same procedure for elements with class name site. For URL validation, another regular expression has been used.

If all validations pass, we show the message Validation OK to the user.

References for regular expressions

You can refer to the below mentioned links for further study of regular expressions:

http://www.regular-expressions.info/
http://en.wikipedia.org/wiki/Regular_expression

Checking for empty fields using jQuery
Validating numbers using jQuery

Wouldn't it be better if we could validate the data as soon as the user starts typing? We will not have to wait until the button is clicked and this will be quite informative for the user too.

This recipe is a major enhancement on previous recipes and will show you how you can use live validation in your forms. Users will be notified of errors as they are inputting data in a field.

Create a folder named Recipe6 inside the Chapter5 directory.

1. Create a new file inside Recipe6 folder and name it as index.html. Write the HTML that will create two panels, one for Personal details and the other for Other details. Textboxes of the first panel will have class name required assigned to them. Similarly, the second panel textboxes will have class names required and number assigned to them.

```
<html>
 <head>
  <title>Live validation</title>
  <style type="text/css">
   body{font-family:"Trebuchet MS",verdana;width:450px;}
   .error{ color:red; }
   #info{color:#008000;font-weight:bold; }
  </style>
 </head>
 <body>
  <form action="process.php" method="post">
   <fieldset>
    <legend><strong>Personal</strong></legend>
    <table>
     <tbody>
      <tr>
       <td>Name:* </td>
       <td><input type="text" class="required" /></td>
      </tr>
      <tr>
       <td>Address:* </td>
       <td><input type="text" class="required"/></td>
      </tr>
      <tr>
       <td>Country:* </td>
       <td><input type="text" class="required"/></td>
      </tr>
     </tbody>
    </table>
   </fieldset>
   <fieldset>
    <legend><strong>Other Details</strong></legend>
    <table>
     <tbody>
      <tr>
      <tr>
       <td>Age:* </td>
       <td><input type="text" class="required number"/>
       </td>
      </tr>
      <tr>
       <td>Monthly Expenses:* </td>
       <td><input type="text" class="required number"/>
       </td>
      </tr>
      </tr>
     </tbody>
    </table>
   </fieldset>
```

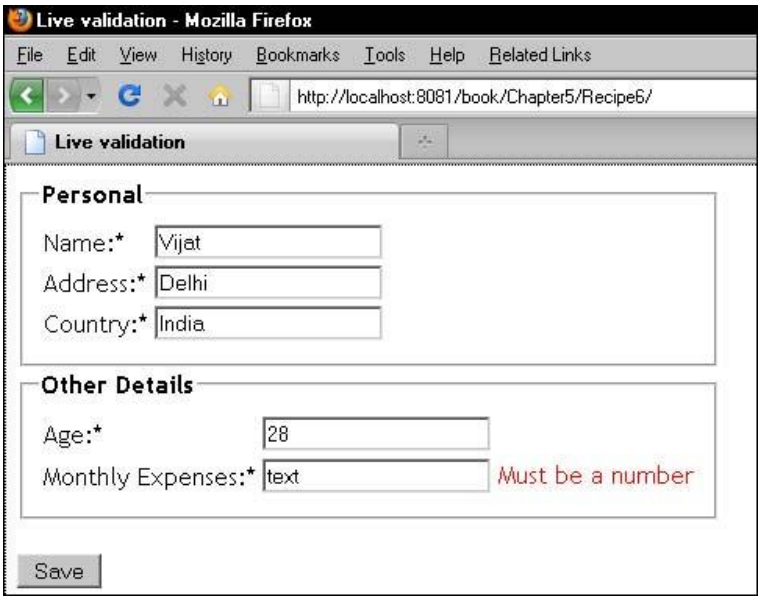
```
    <span id="info"></span>
    <br/>
    <input type="button" value="Save" id="save" />
   </form>
  </body>
</html>
```

2. To bring our form to life, include the jQuery library first. Then write an event handler for textboxes that will execute when any of the textboxes gets focus or a key is released in any of the textboxes. This code will execute as the user is typing and will show an error message on a failed validation condition. Finally, add an event handler for the Check button also because the user might click on the Check button without entering any data in the form.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('input:text').bind('focus keyup',validate);

  function validate()
  {
   var cur = $(this);
   cur.next().remove();
   if(cur.hasClass('required'))
   {
    if ($.trim(cur.val()) == '')
    {
     cur.after('<span class="error"> Mandatory field
            </span>');
     cur.data('valid', false);
    }
    else
    {
     cur.data('valid', true);
    }
   }

   if(cur.hasClass('number'))
   {
    if (isNaN(cur.val()))
    {
     cur.after('<span class="error"> Must be a number
            </span>');
     cur.data('valid', false);
    }
    else
    {
     dataValid = true;
     cur.data('valid', true);
    }
   }
  }

  $('#save').click(function()
```

```
{
  var dataValid = true;
  $('.required').each(function()
  {
    var current = $(this);
    if(current.data('valid') != true)
    {
      dataValid = false;
    }
  });

  $('.number').each(function()
  {
    var current = $(this);
    if(current.data('valid') != true)
    {
      dataValid = false;
    }
  });

  if(dataValid)
    $('#info').html('Validation OK');
  else
    $('#info').html('Please fill correct values in fields.');
 });
});
</script>
```

The output should look similar to the following screenshot on a failed validation:



Since we are going to validate all the fields instantly as user types we attach two event handlers to the textboxes—focus and keyup. keyup will execute when the user releases a key on the keyboard and focus will execute when the

user places the cursor in a textbox either by clicking it through a mouse or by using the *Tab* key. Both event handlers will call the same validate function. This way we will be able to validate the value as soon as it is entered in a textbox.

The validate function will now perform the same functions as we have seen in the last few recipes. It will get the value of textbox and check it for blank values and numeric values, as specified by the class name of the target textbox.

However, there is one problem here. If the user does not fill any values and just clicks on the Save button, we will not be able to detect if any values are filled or not. To resolve this, we will take two steps.

First, while validating in the validate function, we will save a value true or false for each textbox. This will be done by using the data() method of jQuery that stores data with DOM elements. If a field validates we save the value with key valid to it. The value against the key will be either true or false.

There is also an event handler attached to the Save button. Now suppose the user clicks the Save button without doing anything with the textboxes. We then select the textboxes and check if there is data associated with the textboxes or not. The key name should be valid and its value should be true. If we do not get a value true, it means the fields have not been validated yet and we set the variable dataValid to false. We then repeat the same process with textboxes and with the CSS class number. Finally, we show a message to the user depending on the value of the dataValid variable.

Checking for empty fields using jQuery

Validating numbers using jQuery

Validating e-mail and website addresses using regular expressions

Strengthening validation: validating again in PHP

As mentioned previously, client-side validation should always be accompanied by server-side validation. If users turn off JavaScript on their browser and there is no server-side validation, then they can enter whatever they want. This could lead to disastrous results like your database being compromised and so on.

This recipe will go through the validation methods and functions available in PHP, which we can use to validate the data.

Create a new folder named Recipe7 inside the Chapter5 directory .

Make sure your version of PHP is >5.2. We will be using filter functions that are available only after PHP >=5.2

1. Create a file named index.php inside the newly-created Recipe7 folder. Create a form with different type of fields for entering strings, numbers, e-mail addresses, and website addresses.

```
<html>
 <head>
  <title>Server Side validation</title>
  <style type="text/css">
   body{font-family:"Trebuchet MS",verdana;width:450px;}
   .error{ color:red; }
   .info{color:#008000;font-weight:bold; }
  </style>
 </head>
 <body>
  <form method="post">
   <fieldset>
    <legend><strong>Information form</strong>
      (All fields are mandatory)</legend>
    <table>
     <tbody>
      <tr>
       <td>Name: </td>
       <td><input type="text" name="userName"/></td>
      </tr>
      <tr>
       <td>Address: </td>
       <td><input type="text" name="address"/></td>
      </tr>
      <tr>
      <tr>
       <td>Age: </td>
       <td><input type="text" name="age"/></td>
      </tr>
      <tr>
       <td>Mail: </td>
       <td><input type="text" name="email"/></td>
      </tr>
      <tr>
       <td>Website: </td>
       <td><input type="text" name="website"/></td>
      </tr>
      </tr>
     </tbody>
    </table>
   </fieldset>
   <br/>
   <input type="submit" name="save" value="Submit"/>
  </form>
 </body>
</html>
```

The form should look similar to the following screenshot:

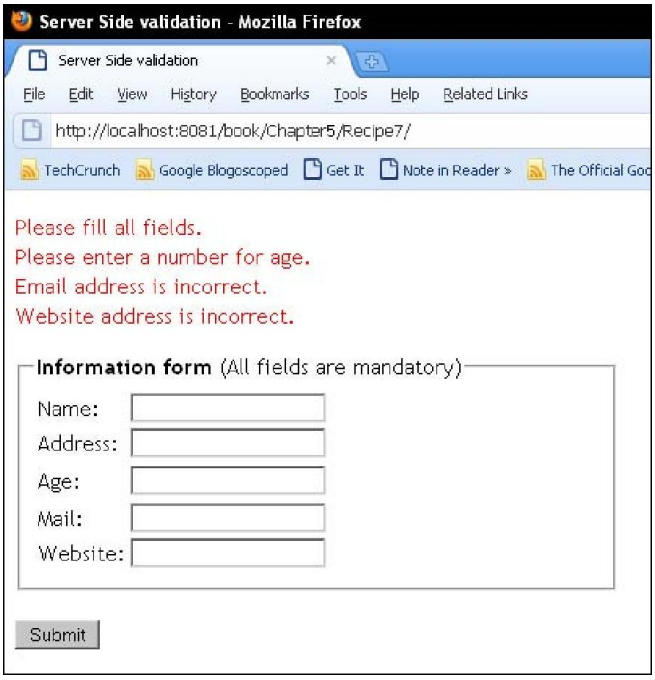
2. When the form is submitted, it will go to the index.php file. Hence, we will place our validations at the beginning of this file. Shown below is the PHP code that needs to be placed at the beginning of the index.php file. This code checks all the fields and upon finding any error it pushes an error message into an array.

```
<?php
 if(isset($_POST['save']))
 {
  $name = trim($_POST['userName']);
  $address = trim($_POST['address']);
  $age = trim($_POST['age']);
  $email = trim($_POST['email']);
  $website = trim($_POST['website']);

  $errorArray = array();
  if($name == '' || $address == '' || $age == '' || $email == ''
            || $website == '')
  {
   array_push($errorArray, 'Please fill all fields.');
  }
  if(filter_var($age, FILTER_VALIDATE_INT) == FALSE)
  {
   array_push($errorArray, 'Please enter a number for age.');

  }
  if(filter_var($email, FILTER_VALIDATE_EMAIL) == FALSE)
  {
   array_push($errorArray, 'Email address is incorrect.');
  }
  if(filter_var($website, FILTER_VALIDATE_URL) == FALSE)
  {
   array_push($errorArray, 'Website address is incorrect.');
  }
 }
?>
```

3. As you can see in the previous code, we are creating an array of error messages (if any). The following code will print these error messages on the browser. Place this code just after the <form> tag opens:

```
<?php
    if(count($errorArray) > 0)
    {
?>
    <p class="error">
<?php
      foreach($errorArray as $error)
      {
        echo $error.'<br/>';
      }
?>
    </p>
<?php
    }
?>
```

4. Open your browser and point it to the index.php file. Enter some incorrect values in the form and click on the Submit button. You will see error messages in the form of a list in your browser.



First, we confirm the form submission using the isset function for $_POST['save']. Then, we collect the values of all form variables in separate variables. Next, we declare an array $errorArray that will collect all the error messages. After that, we check if the fields
are blank or not. If any of the field is found blank, we push an error message in the
$errorArray array.

Next comes the use of PHP's filter_var() function. This function takes three parameters out of which the last two

are optional. The first parameter is the value that is to be filtered. The second parameter is the ID of the Validate filter that defines the type of validation to be done. For example, FILTER_VALIDATE_INT validates the value as integer. In the previous example, we have used three of them, FILTER_VALIDATE_INT, FILTER_VALIDATE_EMAIL, and FILTER_VALIDATE_URL.

filter_var() returns the filtered value on success, and false on failure. In the previous code if we encounter a false value, we push a related error message to the $errorArray array.

Then in the form we check the count for $errorArray. If the number of elements in this array is not equal to zero, then there is some error. So, we iterate in this array and print all the error messages.

List of Validate filters

FILTER_VALIDATE_INT
FILTER_VALIDATE_FLOAT
FILTER_VALIDATE_EMAIL
FILTER_VALIDATE_URL
FILTER_VALIDATE_BOOLEAN
FILTER_VALIDATE_REGEXP
FILTER_VALIDATE_IP

To see the list of all Validate filters available in PHP, you can refer to this URL from the PHP site: http://www.php.net/manual/en/filter.filters.validate.php.

Sanitizing data

Apart from validation filter_var() can also be used to sanitize the data. Data sanitizing refers to removing any malicious or undesired data from the user's input. The syntax remains the same, the only difference is that instead of passing Validate filters as the second parameter, Sanitize filters are passed. Here are some commonly-used Sanitize filters:

FILTER_SANITIZE_EMAIL
FILTER_SANITIZE_NUMBER_FLOAT
FILTER_SANITIZE_NUMBER_INT
FILTER_SANITIZE_SPECIAL_CHARS
FILTER_SANITIZE_STRING
FILTER_SANITIZE_URL
FILTER_SANITIZE_ENCODED

A list of all Sanitize filters can be found on the PHP website at this URL : http://www.php.net/manual/en/filter.filters.sanitize.php

Validating numbers using jQuery
Validating e-mail and website addresses using regular expressions
Displaying errors as user types: performing live validation

We will create an example where users will be able to vote for their favorite browsers. Once voted, they will not be able to vote for another day, that is 24 hours. Votes will be stored in an XML file. We will also display the votes in a nice graphical format.

XML file has been used just for the example. In real world applications, data will be loaded from databases or web services (which can return anything like XML, JSON, or any other format).

Create a folder named Recipe8 inside the Chapter5 directory.

1. OK. This recipe is going to be a bit long, so grab a mug of coffee and start. First of all, create an XML file in the Recipe8 folder and name it as browsers.xml. This file will have information about the browsers that we will display to the user.

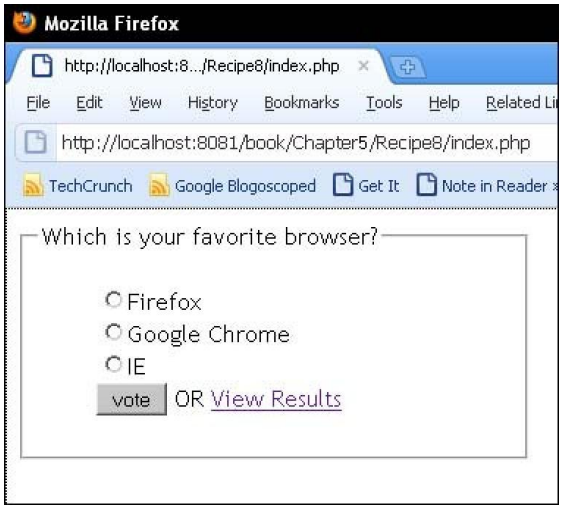
```
<?xml version="1.0"?>
<browsers>
 <browser name="Firefox" value="FF" votes="200"/>
 <browser name="Google Chrome" value="GC" votes="130"/>
 <browser name="IE" value="IE" votes="30"/>
</browsers>
```

2. Now create a PHP file named index.php. We will read the XML file and present the user a list of browsers to select from. This file also contains the code that will handle form submission. Also, the user will not be able to vote more than once in a day.

```
<?php
if(isset($_POST['vote']))
{
 if(isset($_COOKIE["voted"]))
 {
  $message = 'You have already voted. You cannot vote more than
          once per day.';
 }
 else
 {
  $message = 'Your vote has been saved';
  $dom = new DOMDocument();
  $dom->load('browsers.xml');
  $xpath = new DomXPath($dom);
  $units = $xpath->query('//browser');

  foreach ($units as $unit)
  {
   $value = $unit->getAttribute('value');
   if($value == $_POST['browser'])
   {
    $votes = $unit->getAttribute('votes');
    $unit->setAttribute('votes', ++$votes);
```

```
      setcookie("voted", true, time()+ (24*60*60));  /* expire
         in 24 hours */
      break;
     }

   }
   $dom->save('browsers.xml');
 }
}
?>
<html>
 <head>
  <title></title>
  <style type="text/css">
  body{font-family:"Trebuchet MS",verdana;width:350px;}
  ul{list-style:none;}
  </style>
 </head>
 <body>
  <form method="post">
   <fieldset>
    <legend>Which is your favorite browser?</legend>
    <ul>
  <?php
     $dom = new DOMDocument();
     $dom->load('browsers.xml');
     $xpath = new DomXPath($dom);
     $browsers = $xpath->query('//browser');

     foreach ($browsers as $browser)
     {
      $checked = $_POST['browser'] ==
            $browser->getAttribute('value')? 'checked': '';
      echo '<li><input type="radio" '.$checked.' name="browser" value="'.$browser-
>getAttribute('value').'">'.$browser->getAttribute('name').'</li>';
     }
  ?>
      <li style="color:red;"><?php echo $message; ?></li>
      <li><input type="submit" name="vote" value="vote" /> OR <a href="results.php" id="results">View
Results</a></li>
    </ul>
   </fieldset>
  </form>
 </body>
</html>
```

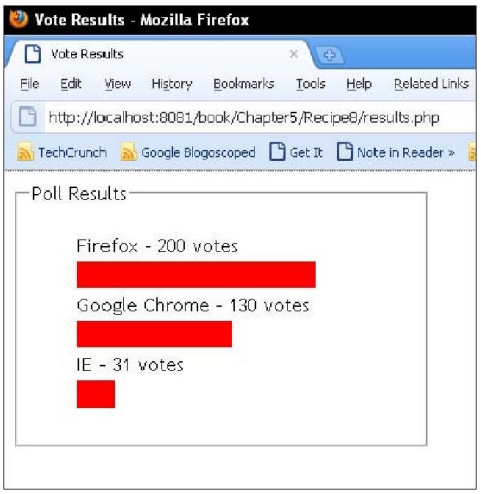
3. Run the file in the browser and you will see some radio buttons and a vote button as shown in the following screenshot:



4. The above page also contains a link to view the results. To create that page, open a new file and save it as results.php. The code in this file will read the XML file and will display votes for each browser.

```
<html>
 <head>
  <title>Vote Results</title>
  <style type="text/css">
  body{font-family:"Trebuchet MS",verdana;width:350px;}
  ul{list-style:none;}
  li{height:25px;}
span{background-color:red;color:#fff;float:left;}
  </style>
 </head>
 <body>
  <fieldset>
   <legend>Poll Results</legend>
<?php
 $dom = new DOMDocument();
 $dom->load('browsers.xml');
 $xpath = new DomXPath($dom);
 $browsers = $xpath->query('//browser');

 echo '<ul  >';

 foreach ($browsers as $browser)
 {
  $name = $browser->getAttribute('name');
  $votes = $browser->getAttribute('votes');
  echo '<li>'.$name.' - '.$votes. ' votes</li>';
  echo '<li><span style="width:'.$votes.'px;"> </span></li>';style="width:'.$style="width:'.$
 }
 echo '</ul>';

?>
  </fieldset>
 </body>
</html>
```

5. All done and we are ready to run our example now. Run the index.php file in your browser and you will see the form. Select the last radio button and click on the vote button. You will see a message that says Your vote has been saved. Now select a browser and click the vote button again. This time you will see an error message
   that says You have already voted. You cannot vote more than once per day.
6. Now click on the View Results link. This will open a new page and you will see the number of votes for each browser. Your vote will increase the vote count for IE from 30 to 31.



Let us start by examining the structure of the browsers.xml file. This XML contains three browser nodes, each defining one browser. Each node has three attributes: name, value, and nodes. Name will be displayed to the user, value will be used in internal processing, and votes are the number of votes for each browser.

Coming to the index.php file now, we will start from the HTML. Using the DOM Document functions we load the XML file and create an unordered list from it. A radio button is created for each browser. In the end, a button is created for vote and another link for View Results.

Now here's a summary of what happens after a form is submitted.

To find out if a user has previously voted or not, we check the Superglobal $_COOKIE. If this cookie contains an entry for voted, this means the user has voted previously and we show an error message.

If the user has not voted already, we increase one vote in the XML file.

To add a vote for the selected browser, we load the XML using DOMDocument.

Then we search through all browser nodes and check the attribute value against the value selected by the user. This value is available in $_POST['browser'].

Once a match is found we increase the number of votes by one against that browser.

Then we set a cookie named voted, which will sit on the user's browser. PHP's setcookie function is used to set the cookie and it is set to expire after 24 hours. This will prevent the user from voting more than once in a single day.

Finally, save the XML using the save method of DOM.

To generate the page results.php, load the XML file using DOM Document again and iterate through all browser nodes. We create an unordered list again. For each browser, two list items are created. In the first li, we write the name of browser and number of votes cast against it. The second li creates a span element with its width set to the number of votes in pixels. This will create the effect of a bar chart.

Cookie expiration time

In the previous example, we have set the cookie to expire after a day. You can change this as per your requirements. Just note that it is passed as a UNIX timestamp and hence you will have to pass it in seconds.

Reading an XML using DOM extension in Chapter 3

Modifying an XML using DOM extension in Chapter 3

While a user is filling out some form, you may want to restrict the HTML tags that are allowed through user input. Some unwanted tags like <script> tags can cause potential harm to your site and its data.

This recipe will teach you how to filter the tags from data entered in a web form and accept only specific tags.

Create a folder named Recipe9 inside the Chapter5 directory.

1. Create a new file and save it as index.html. Now, create two textarea elements and a button. The first textarea is where the user will enter the text in HTML format. The second textarea will show the HTML after disallowed tags are stripped from it.

```
<html>
 <head>
  <title>Strip tags</title>
  <style type="text/css">
   body{font-family:"Trebuchet MS",verdana;width:700px;}
  </style>
 </head>
 <body>
  <form>
   <table>
    <tr>
     <td valign="top">Write some HTML in the box<br/>
     (Only allowed HTML tags are<br/>&lt;b&gt;,&lt;u&gt;,&lt;
     i&gt; and &lt;strong&gt;.<br/>Other tags will
     be removed):</td><td>
      <textarea id="comment" cols="50" rows="10"></textarea>
     </td>
    </tr>
    <tr>
     <td valign="top">This is how your HTML will look:</td>
     <td>
      <textarea id="stripped" cols="50" rows="10">
      </textarea>
     </td>
    </tr>
   </table>
```

```
    <input type="button" value="Check" id="check" />
   </form>
  </body>
</html>
```

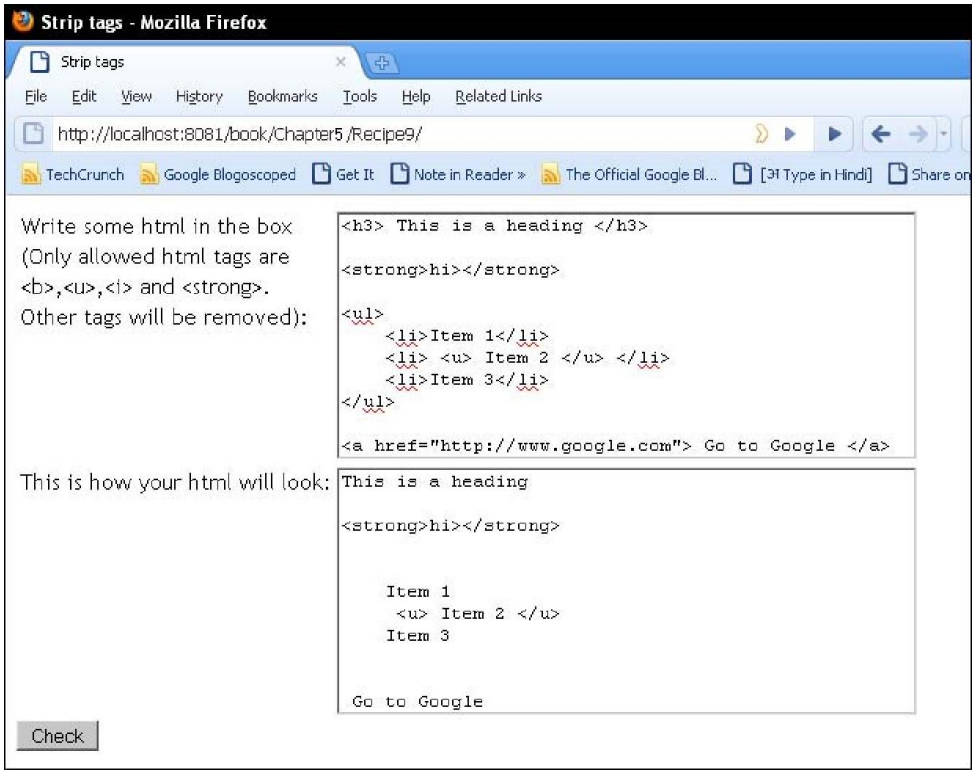
2. Include the jquery.js file and then add an event handler for the Check button. Clicking on this button will send the data of the first textarea to a PHP file, validate.php. On receiving a response, it will be set inside the second textarea.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function ()
 {
  $('#check').click(function()
  {
   $.post(
    "validate.php",
    { comment: $('#comment').val() },
    function(data)
    {
     $('#stripped').val(data);
    });
  });
 });
</script>
```

3. Create another file for the PHP code and name it validate.php. The code in this file will strip the disallowed HTML tags from the input data and will echo them back to the browser.

```
<?php
$text = $_POST['comment'];
echo trim(strip_tags($text, '<b><u><i><strong>'));
?>
```

4. Now, open your browser and run the index.html file. Write some HTML in it and click on the Check button. The second textarea will show the HTML after disallowed tags are stripped from it.



On clicking the Check button, an AJAX request is sent to the PHP file validate.php. Here comes the main part. We get the data received from the POST request. Then we use the PHP function strip_tags(). This function removes the HTML tags from the input string. The first parameter to this function is the input string that we need to strip tags from. The second parameter is optional. If not passed, this function will strip all HTML tags from the input string. In our example, we want to allow four tags: <b>,<u>,<i>, and , therefore we passed these as second parameters. The function will now remove all HTML tags from the input string except these four. It returns the resulting string that you can now safely save to a database or perform other operations on. In this example, we echo it to the browser to see how it will look. On the browser, jQuery inserts it into the second textarea.

PHP tags are stripped too

Any HTML comments in input string and PHP tags are automatically stripped.

6

Data Binding with PHP and jQuery

In this chapter, we will cover:

- Fetching data from a database and displaying it in a table format
- Collecting data from a form and saving it to a database (Registration form)
- Filling chained combo boxes that depend upon each other
- Checking username availability from a database
- Paginating data for large record sets
- Adding auto suggest functionality to a textbox
- Creating a tag cloud

This chapter will explain some recipes where we will use a database along with PHP on the server side. A database is an essential part of almost every dynamic web application. PHP provides a large number of functions to interact with the database. The most commonly used database along with PHP is MySQL. In this chapter, we will be using another version of MySQL called MySQLi or MySQL improved. It provides significant advantages over the MySQL extension; most important of them being the support for the object-oriented interface as well as the procedural interface. Other features include support for transactions, prepared statements, and so on.

You can read more about MySQLi on the PHP site at http://www.php.net/manual/en/book.mysqli.php.

MySQLi extension is available with PHP version 5.0 or higher. So, make sure you have the required PHP version. If you are running PHP 5 or a higher version, you will have to configure MySQL separately as a default PHP support, for MySQL was dropped starting from PHP versions 5.0 and higher.

Cleaning data before use

Throughout the recipes in this book, we have used user input directly by pulling these from $_GET or $_POST arrays. Although this is okay for examples, in practical websites and applications, user data must be properly cleaned and sanitized before performing any operations on it to make your application safe from malicious users. Below are some links where you can get more information on how to make your data safe, and security in general.

PHP Security Consortium: http://phpsec.org/

PHP Manual: http://php.net/manual/en/security.php

This is a simple recipe where we will get some data from a table and we'll display it in a page. Users will be presented with a select box with options to choose a programming language. Selecting a language will get some functions and their details from the database.

Create a new folder named Recipe1 inside the Chapter8 directory. Now, using phpMyAdmin create a table named language in the exampleDB database using the following query.

```
CREATE TABLE `language` (
  `id` int(3) NOT NULL auto_increment,
  `languageName` varchar(50) NOT NULL,
  PRIMARY KEY  (`id`)
);
```

Insert two records for languageName in this table, namely PHP and jQuery. Now, create another table functions that will have function names and details related to a language.

```
CREATE TABLE `functions` (
  `id` int(3) NOT NULL auto_increment,
  `languageId` int(11) NOT NULL,
  `functionName` varchar(64) NOT NULL,
  `summary` varchar(128) NOT NULL,
  `example` text NOT NULL,
  PRIMARY KEY  (`id`)
);
```

languageId is the ID of the language that is in the language table. Now, insert some records in this table using phpMyadmin with some data for PHP and some for jQuery. Here is a snapshot of what the functions table will look like after filling it with data:

id	languageId	functionName	summary	example
1	1	simplexml_load_file	Interprets an XML file into an object	$xml = simplexml_load_file('test.xml'); print_r($...
2	1	array_push	Push one or more elements onto the end of array	$arrPets = array('Dog', 'Cat', 'Fish'); array_pu...
3	1	ucfirst	Make a string's first character uppercase	$message = 'have a nice day; $message = ucfirst($...
4	1	mail	used to send email	$message = "Example message for mail"; if(mail('t...
5	2	$.get	Load data from the server using a HTTP GET request...	$.ajax({ url: url, data: data, success: s...
6	2	hover	hover method accepts 2 functions as parameters whi...	$(selector).hover(function() { //executes on m...
7	2	bind	Attach a handler to an event for the elements	$(element).bind('click', function() { alert('...
8	2	jQuery.data	Store arbitrary data associated with the specified...	jQuery.data(element, key, value);

1. Create a file named index.php in the Recipe1 folder. Using methods of MySQLi class, select data from the language table, and populate a select box with list of languages. Also, create a p element that will show the functions for the selected language.

```
<html>
 <head>
  <style type="text/css">
  body{font-family: "Trebuchet MS", Verdana, Arial;width:600px;}
  div {  background-color: #F5F5DC;  }

  </style>
 </head>
 <body>
<?php
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
if (mysqli_connect_errno())
{
  die('Unable to connect!');
}
else
{
 $query = 'SELECT * FROM language';
 if ($result = $mysqli->query($query))
 {
  if ($result->num_rows > 0)
  {
?>
    <p>
     Select a language
     <select id="selectLanguage">
      <option value="">select</option>
<?php
```

```
    while($row = $result->fetch_assoc())
    {
?>
     <option value="<?php echo $row[0]; ?>"><?php echo $row[1]; ?></option>
<?php
    }
?>
     </select>
   </p>
   <p id="result"></p>
<?php
   }
   else
   {
    echo 'No records found!';
   }
   $result->close();
 }
 else
 {
   echo 'Error in query: $query. '.$mysqli->error;
 }
}
$mysqli->close();
?>
 </body>
</html>
```

2. Now, add a reference to the jQuery file. After this, write the event handler for a select box that will be fired on selecting a value from the combo box. It will send an AJAX request to a PHP file results.php, which will get the data for the selected language and will insert it into the p element.

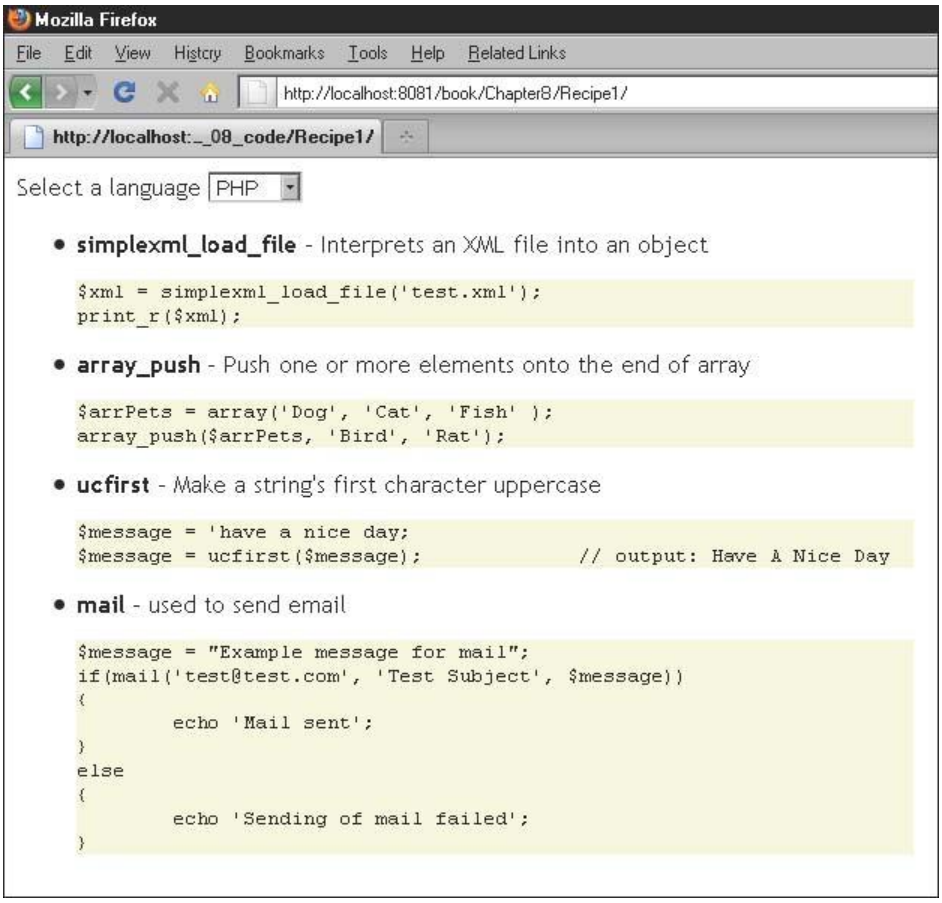
```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
   $('#selectLanguage').change(function()
   {
    if($(this).val() == '')  return;
    $.get(
      'results.php',
      { id : $(this).val() },
      function(data)
      {
        $('#result').html(data);
      }
    );
   });
 });
</script>
```

3. Create another results.php file that will connect to the database exampleDB and will get data specific to a language from the database. It will then create the formatted HTML from the results and will send it back to the browser where jQuery inserts it into the p element.

```
<?php
```

```
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
$resultStr = '';
$query = 'SELECT functionName, summary, example FROM functions where languageId='.$_GET['id'];
if ($result = $mysqli->query($query))
{
 if ($result->num_rows > 0)
 {
  $resultStr.='<ul>';
  while($row = $result->fetch_assoc())
  {
   $resultStr.= '<li><strong>'.$row['functionName'].'</strong>
   - '.$row['summary'];
   $resultStr.= '<div><pre>'.$row['example'].'</pre></div>';
   '</li>';
  }
  $resultStr.= '</ul>';
 }
 else
 {
  $resultStr = 'Nothing found';
 }
}
echo $resultStr;
?>
```

4. Now, run the index.php file in the browser and you will see a combo box with two options: PHP and jQuery. Select any option and you will see the results in the form of a bulleted list.



First, we create a new object of MySQLi class using its constructor. We pass the host, database user name, password, and database name to it. Then, we check for errors, if any, while connecting to the database. In case of an error, we display an error message and terminate the script.

Then, we use the query method of the mysqli class to select all data from the language table. If the query is successful we get the result object in the $result variable. The $result variable that we have is an object of the MySQLi_Result class. The MySQLi_Result class provides several methods to extract data from the object. We have used one such method called fetch_assoc()that fetches a row as an associative array. Using a while loop, we can iterate in the $result object one row at a time. Here, we create a select box with ID selectLanguage and fill the language names as its options and languageId as values for the options.

In jQuery code, we have an event handler for the change event of the combo box. It takes the value of the select box and sends it to the results.php file, using a GET AJAX request.

The results.php file connects to the exampleDB database and then writes a query for selecting data for a particular language. jQuery sends an id parameter with an AJAX request that will be used in the query. Like the index.php page, we get the results in the $result variable. Now, we iterate over this result and create an unordered list and assign it to the $resultStr variable. Each list item contains a function name, a brief description about it, and an example. In case of any error, the variable $resultStr is assigned an error message.

Finally, we echo the $resultStr variable received by jQuery. jQuery then inserts the received HTML in the p element with ID result.

What is a constructor?

In object-oriented programming, a constructor is a method that is invoked whenever a new object of that class is created. A constructor has the same name as the class name.

```
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
```

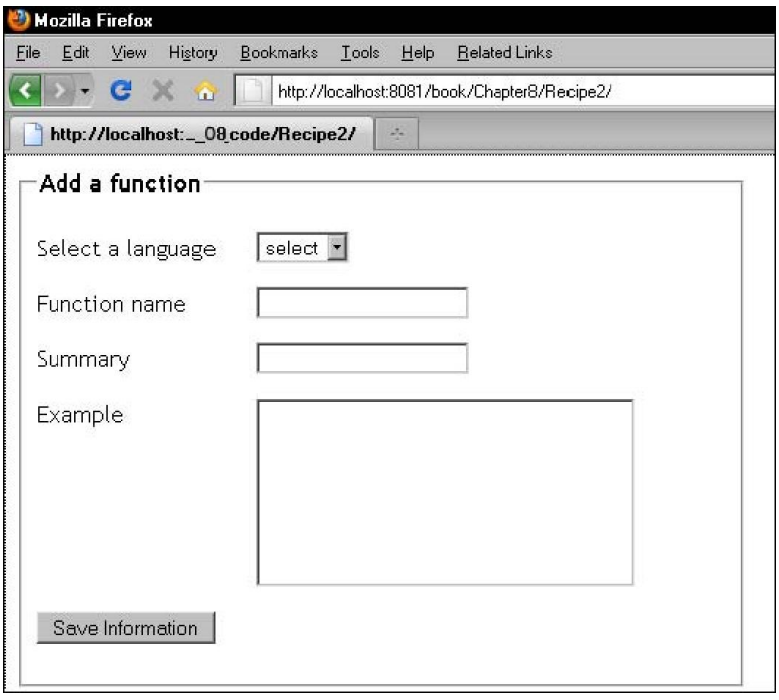
The above line creates a new object of mysqli class, which has a constructor that takes four arguments.

One thing to keep in mind is: in PHP5 and above versions, a constructor is defined as __construct() whereas in prior versions the constructor has the same name as the class name. To read more about constructors in PHP refer to the PHP site: http://www.php.net/manual/en/language.oop5.decon.php

Using the same two tables of the previous recipe, we will create a form that will allow the user to select a language, add a function name, its summary, and related examples. We will then save this information to the functions table with the selected language.

Create Recipe2 folder inside the Chapter8 directory.

1. Create a file named index.php inside the Recipe2 folder. Now, create a form with four fields. First, create a select box and query the language table to fill languages in it. Next, create two textboxes for Function name and Summary. Finally, create a textarea in which users will enter the example for that function. Assign a CSS class named required to each of these elements.



2. Before the closing of body tag, include the jquery.js file and after that, write the event handler function for the form's submit event. This function will perform a basic validation by checking each element's value. If any of the fields is blank, it will display an error message. If there are no errors, the form will be submitted.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  $('#frmMain').submit(function()
  {
   var flag = true;
   $('#error').empty();
   $('.required').each(function()
   {
    if(jQuery.trim($(this).val()) == '' )
    {
     flag = false;
    }
   });
   if(!flag)
   {
    $('#error').html('Please fill all the fields');
    return false;
   }
   else
   {
    return true;
   }
  });
 });
</script>
```

3. Now, when the form is submitted, PHP will take the values for each element from the global $_POST array and will assign them to different variables, after escaping them. Then an INSERT query will execute and will insert these values into the database. An appropriate message will be displayed, depending on whether the query has succeeded or failed. Below is the full code for the index.php file.

```
<html>
 <head>
  <style type="text/css">
  body{ font-family: "Trebuchet MS", Verdana, Arial;
       width:500px; }
  input,textarea { vertical-align:top; }
  label{ float:left; width:150px;}
  </style>
 </head>
 <body>
<?php
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
if(isset($_POST['save']))
{
 $language = $mysqli->real_escape_string($_POST['language']);
 $functionName = $mysqli->real_escape_string($_POST['functionName']);
 $summary = $mysqli->real_escape_string($_POST['summary']);
```

```
 $example = $mysqli->real_escape_string($_POST['example']);
 $query = 'INSERT INTO functions (
    languageId ,
    functionName ,
    summary ,
    example
    )
    VALUES ('.$language.', "'.$functionName.'", "'.$summary.'","'.$example.'")';
     if ($mysqli->query($query))
 {
  echo 'Data Saved Successfully.';
 }
 else
 {
  echo 'Cannot save data.';
 }

}
$query = 'SELECT * FROM language';
if ($result = $mysqli->query($query))
{
 if ($result->num_rows > 0)
 {
?>
   <fieldset>
   <legend><strong>Add a function</strong></legend>
   <form action="" method="post" id="frmMain">
    <p>
     <label>Select a language</label>
      <select name="language" class="required">
       <option value="">select</option>
<?php
        while($row = $result->fetch_array())
        {
?>
         <option value="<?php echo $row[0]; ?>">
          <?php echo $row[1]; ?></option>
<?php
        }
?>
      </select>
    </p>
    <p>
     <label>Function name </label>
     <input type="text" name="functionName"  class="required"/>
    </p>
    <p>
     <label>Summary</label>
      <input type="text" name="summary" class="required"/>
    </p>
    <p>
     <label>Example</label> <textarea rows="10" cols="30"
      name="example" class="required"></textarea>
    </p>
```

```
    <p>
      <strong id="error"></strong>
    </p>
    <p>
      <input type="submit" name="save"
        value="Save  Information"/>
    </p>
   </form>
   </fieldset>
<?php
 }
 else
 {
   echo 'No records found!';
 }
 $result->close();
}
else
{
 echo 'Error in query: $query. '.$mysqli->error;
}

$mysqli->close();
?>
 </body>
</html>
```

4. Now, run the file in your browser and fill some values in the form. Click on the Save Information button and it will save the values to the functions table in the database. You will also see a message Data Saved Successfully on successful execution of the query. Leaving any fields blank and trying to submit the form will display an error message.

First, we connect to the database using the constructor of mysqli class. Next, the if statement checks whether the form has been submitted or not. Hence, this part will be executed after the form submission. We will look into this part in detail later in this chapter.

```
if(isset($_POST['save']))
{
}
```

Outside the above condition, we query the language table using a SELECT statement that gets us the languages from the database. We then fill these languages and their values inside the select box. Other fields include two textboxes and a textarea.

After the form is submitted with non-blank values, PHP fetches these values from the $_POST Superglobal and escapes it using real_escape_string() method of mysqli class. This function escapes the user data so that it is ready to be used in a query. Then, we insert the values for the language, function name, and example using an INSERT query. query() will return true on success and false on failure. We then display the final message to the user based on this return value.

real_escape_string() function

The real_escape_string() function is used to escape special characters in a string. SQL queries may throw an error if the data present in them is not escaped properly. You should always use it in your database queries.

Also note that you need to be connected to a database to be able to use this function.

Return values for mysqli->query()

For statements such as SELECT, SHOW, and so on, this method returns an object of class MySQLi_Result. For statements like INSERT, UPDATE, and DELETE, it returns either
TRUE or FALSE.

Checking for empty fields using jQuery in Chapter 5

This recipe tries to solve a very common task that is seen in many web applications, that is, filtering contents of a combo box according to the selection made in its previous combo box.

We will create an example where the user will be presented with three select boxes—one each for country, state, and town. Selecting a country will get its states and selecting a state will get
its towns. Finally, on selecting a town we will display some information related to it.

The most important point here is that there will not be any page reloads. Instead, we will use AJAX to filter the contents silently. This will create a better user experience compared to classic web application behavior where it would have required a full-page reload on each selection.

Create a folder named Recipe3 inside the Chapter8 directory. Now, we will require four tables in our database. Once again, open phpMyadmin, create these four tables, and fill
them with the desired values.

Country

```
CREATE TABLE `country` (
  `id` int(11) NOT NULL auto_increment,
  `countryName` varchar(64) NOT NULL,
  PRIMARY KEY  (`id`)
);

INSERT INTO `country` (`id`, `countryName`) VALUES
(1, 'India');
```

States

```
CREATE TABLE `states` (
  `id` int(11) NOT NULL auto_increment,
  `countryId` int(11) NOT NULL,
  `stateName` varchar(64) NOT NULL,
  PRIMARY KEY  (`id`)
);

INSERT INTO `states` (`id`, `countryId`, `stateName`) VALUES
```

```
(1, 1, 'U.P.'),
(2, 1, 'Uttarakhand');
```

Towns

```
CREATE TABLE `towns` (
  `id` int(11) NOT NULL auto_increment,
  `stateId` int(11) NOT NULL,
  `townName` varchar(64) NOT NULL,
  PRIMARY KEY  (`id`)
);

INSERT INTO `towns` (`id`, `stateId`, `townName`) VALUES
(1, 1, 'Lucknow'),
(2, 1, 'Bareilly'),
(3, 2, 'Pithoragarh'),
(4, 2, 'Dehradun'),
(5, 2, 'Nainital');
```

Towninfo

```
CREATE TABLE `towninfo` (
  `id` int(11) NOT NULL auto_increment,
  `townId` int(11) NOT NULL,
  `description` text NOT NULL,
  PRIMARY KEY  (`id`)
);

INSERT INTO `towninfo` (`id`, `townId`, `description`) VALUES
(1, 3, 'Pithoragarh is a beautiful town situated in Kumaon region of Uttarakhand. It has an average elevation of 1,514 metres (4,967 feet) above sea level.'),
(2, 4, 'Dehradun also known as Doon is the capital city of Uttarakhand. It is around 250 Kilometers from national capital Delhi.\r\nRice and Lychee are major products of this city.'),
(3, 1, 'Lucknow is the capital city of U.P. or Uttar Pradesh.\r\nLucknow has Asia''s first human DNA bank.\r\nIt is popularly known as The City of Nawabs, Golden City of the East and The Constantinople of India.');
```

1. Create a file index.html inside the Recipe3 folder. Create three combo boxes for country, state, town, and a p element that will display the information about the selected town. Also write some CSS styles in head section for styling these elements. All values in these combo boxes will be filled using AJAX requests.

```
<html>
 <head>
  <style type="text/css">
  body{font-family: "Trebuchet MS", Verdana, Arial;width:600px;}
  ul { list-style:none;margin:0pt;padding:0pt;width:525px;
      float:left; }
  li{ float:left;padding:10px;}
  p{border:1px solid #000; float:left;height:100px;width:500px;}
  select { width:100px;}
  </style>
 </head>
 <body>
  <ul>
   <li>
    <strong>Country</strong>
```

```
    <select id="countryList">
      <option value="">select</option>
    </select>
   </li>
   <li>
    <strong>State</strong>
    <select id="stateList">
      <option value="">select</option>
    </select>
   </li>
   <li>
    <strong>Town</strong>
    <select id="townList">
      <option value="">select</option>
    </select>
   </li>
  </ul>
  <p id="information"></p>
 </body>
</html>
```

2. Before the body tag closes, add the jQuery library. Now, create a function getList that will be called whenever the value in the combo box changes. Depending on which combo box it is, a URL will be set with two parameters: find and id. Finally, an AJAX request will be sent to this URL which will fetch the corresponding results. Function getList() will be called once the document is ready so that we have values available in the Country combo box.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  $('select').change(getList);
  getList();
  function getList()
  {
   var url, target;
    var id = $(this).attr('id');
    var selectedValue = $(this).val();
    switch (id)
    {
```

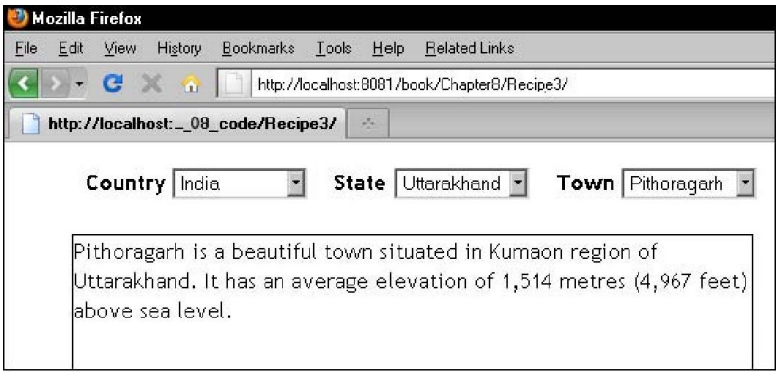
```
    case 'countryList':
      if(selectedValue == '')    return;
      url = 'results.php?find=states&id='+ selectedValue;
      target = 'stateList';
    break;

    case 'stateList':
      if($(this).val() == '')        return;
      url = 'results.php?find=towns&id='+ selectedValue;
      target = 'townList';
    break;

    case 'townList':
      if($(this).val() == '')        return;
      url = 'results.php?find=information&id='+ selectedValue;
      target = 'information';
    break;

    default:
      url = 'results.php?find=country';
      target = 'countryList';
   }
   $.get(
    url,
    { },
    function(data)
    {
     $('#'+target).html(data);
    }
   )
  }

 });
</script>
```

3. The AJAX request will be sent to the results.php file. So, create a new file with this name. This file connects to the database and depending on the values of parameters find and id, it queries the appropriate table and fetches data from it. HTML is generated from this data and is sent back to the browser where jQuery displays it.

```
<?php
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
$find = $_GET['find'];
switch ($find)
{
 case 'country':
   $query = 'SELECT id, countryName FROM country';
 break;
 case 'states':
   $query = 'SELECT id, stateName FROM states WHERE
   countryId='.$_GET['id'];
 break;
 case 'towns':
   $query = 'SELECT id, townName FROM towns
    WHERE stateId='.$_GET['id'];
```

```
 break;
 case 'information':
  $query = 'SELECT id, description FROM towninfo
  WHERE townId='.$_GET['id'] .' LIMIT 1';
 break;
}
if ($mysqli->query($query))
{
 $result = $mysqli->query($query);
 if($find == 'information')
 {
  if($result->num_rows > 0)
  {
   $row = $result->fetch_array();
   echo $row[1];
  }
  else
  {
   echo 'No Information found';
  }
 }
 else
 {
?>
  <option value="">select</option>
<?php
  while($row = $result->fetch_array())
  {
?>
   <option value="<?php echo $row[0]; ?>"><?php echo $row[1]; ?></option>
<?php
  }
 }
}
?>
```

4. Run the index.html file in your browser and you will find the values in the Country combo box. Other boxes will be empty. Select a country and the State combo box will be filled with data. Selecting a state will fill the last combo box (Town). In the end, select a town and an AJAX request will get information related to it and will display
it in the p element.



The HTML code of index.html is almost clear. We have created three combo boxes and a p element. Each element has been assigned an ID: countryList, stateList, townList, and information respectively.

In the jQuery code, we have added a change event listener for all select elements that call a function getList(). getList() defines two variables: URL and target. Then, it gets the ID and the value of the element whose value is changed. Next, is a switch case where the ID of the element is checked in four different cases. If the value from the combo box with ID countryList is selected, we set the find parameter in the URL to states and id parameter as its value. Similarly for stateList box, find is set to towns and for selectbox townList, we set the find parameter to information because on selecting a town we need to show information related to it. In the default case, find is set to country so that it gets all the countries from the database and fills them in first combo box. Along with setting the URL we also set the target element in which data will be inserted.

After the switch case, an AJAX GET request is sent from jQuery to the PHP file results.php. The response received from results.php will be inserted in the target element.

Let's go through the code of results.php now. This script first connects to our exampleDB. Then, we fetch the value of the find key from the $_GET Superglobal. A switch case checks the value of the $find variable and creates a query accordingly. If find is set to states it creates a query to retrieve data from the states table based on countryId. If case is information, it queries the information table for the id of a particular town.

Once the results are retrieved from the database, a while loop is used to iterate over them and a formatted HTML is sent back to the browser where jQuery inserts it into the appropriate target element.

We will write an example of a registration form that will match a user-entered name against all other names in the database and will notify the user whether that username is available or not.

Create a folder for this recipe inside the Chapter8 directory and name it as Recipe4. Open phpMyAdmin and create a new table named users with the following structure and data.

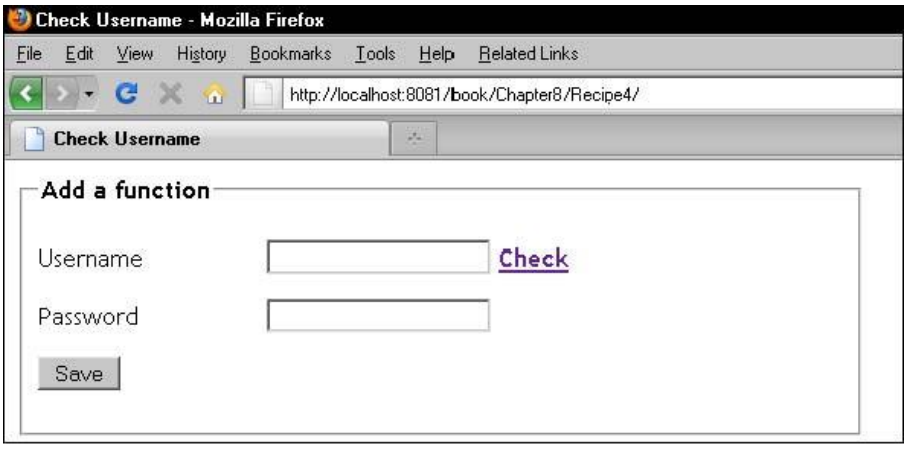
```
CREATE TABLE `users` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(32) NOT NULL,
  `password` varchar(32) NOT NULL,
  PRIMARY KEY (`id`)
);

      INSERT INTO `users` (`id`, `username`, `password`) VALUES
(1, 'holmes', 'sherlockholmes'),
(2, 'watson', 'johnwatson'),
(3, 'sati', 'pranay'),
(4, 'mantu', 'ajayjoshi'),
(5, 'sahji', 'brijsah'),
(6, 'vijay', 'vijayjoshi'),
(7, 'brij', 'brijsah'),
(8, 'arjun', 'samant'),
(9, 'jyotsna', 'sonawane'),
(12, 'ravindra', 'pokharia'),
```

(13, 'prakash', 'joshi'),
(14, 'sahji2', 'aloklal'),

(15, 'basant', 'bhandari')

1. Create a file named index.html in the Recipe4 folder. In this file, create two textboxes for login name and password. Next to the login name, create an anchor that will check the username on clicking it. Another element next to it will show whether that login name is available or not.

```
<html>
 <head>
  <title>Check Username</title>
  <style type="text/css">
  body{ font-family: "Trebuchet MS", Verdana, Arial;
       width:555px; }
  input,textarea { vertical-align:top; }
 label{ float:left; width:150px;}
  #error {font-weight:bold; color:#ff0000;}
  </style>
 </head>
 <body>
  <fieldset>
   <legend><strong>Add a function</strong></legend>
   <form action="" method="post" id="loginForm">
    <p>
     <label>Username </label>
     <input type="text" name="loginName" id="loginName"/>
     <a href="#" id="check"><strong>Check</strong></a>
     <span id="status" style="float:right;"></span>
    </p>
    <p>
     <label>Password</label>
     <input type="password" name="password"/>
    </p>
    <p>
     <span id="error"></span>
    </p>
    <p>
     <input type="submit" value="Save" name="dos"
      id="dosave"/>
    </p>
   </form>
  </fieldset>
 </body>
</html>
```



2. Now include the jquery.js file first. Next, write an event handler function that will be executed when the user clicks on the element with check ID. It will send an AJAX request to the PHP file, check.php, which will return either true or false depending on whether the username is available or not. Another event handler is for the submit event of the form that will allow the form to be submitted when the user has chosen an available username.

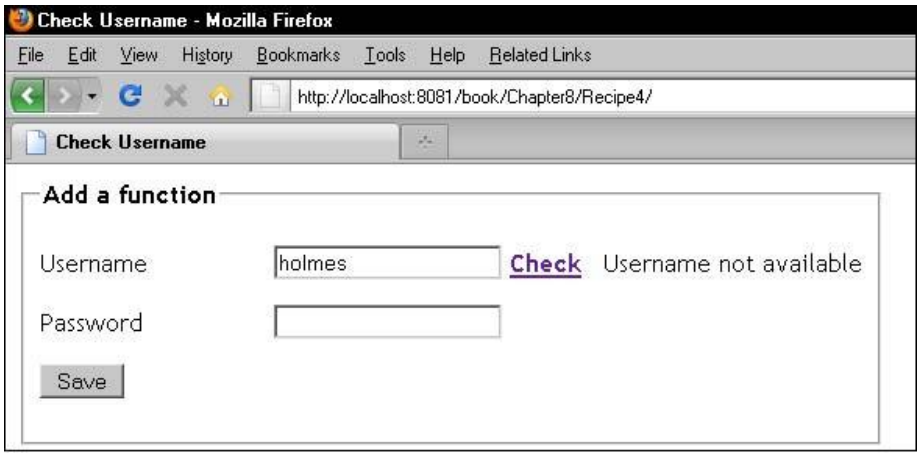
```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  var checked = false;
  $('#check').click(function()
  {
    $('#error').empty();
    var inputValue = $('#loginName').val();
    if(jQuery.trim(inputValue) == ''){return false; }
    $.post(
     'check.php',
     { username : inputValue },
     function(data)
     {
      if(data)
      {
       checked = true;
       $('#status').html('Username is available');
      }
      else
      {
       checked = false;
       $('#status').html('Username not available');
       return false;
      }
     }
    );
  });
  $('#loginForm').submit(function()
  {
   if(checked == false)
   {
```

```
      $('#error').html('Kindly check the username');
      return false;
    }
    else
    {
      return true;
    }
   });
   $('#loginName').focus(function()
   {
    checked == false;
   });
 });
</script>
```

3. Create another file and name it as check.php. This file will check the values supplied by jQuery in the users table and will return true or false.

```
<?php
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
$selectQuery = 'SELECT username as user FROM users WHERE username="'.$_POST['username'].'"';
$result = $mysqli->query($selectQuery);
if($result)
{
 if ($result->num_rows > 0)
 {
   echo false;
 }
 else
 {
   echo true;
 }
}
else
{
 echo false;
}
?>
```

4. Run the index.html file in the browser and enter a username that is already in the database and click on the Check link. You will see an error message Username not available. Entering an available username will show the message Username is available. Trying to submit the form without checking the username will display an error Kindly check the username.



On clicking the Check link an AJAX request is sent to check.php file. This file checks the users table for that username. If there are more than zero records in the table we can be sure that the username is already in use and we return false, otherwise we return true.

jQuery's success callback function checks the value provided by PHP and displays an error message accordingly.

Variable checked is used to prevent the form submission if it takes place without checking a username. Only if a username is available is the variable set to true and the form submission is allowed.

Alternative methods for implementation

In this recipe, we are checking the username on the click of a button. The same check can be implemented on the onkeydown event of the textbox too. This has been left as an exercise for you.

It is best to break down a long list into separate pages and navigate them with buttons such as Previous, Next, and specific page numbers. In this recipe, we will take a long list of HTML elements and will paginate them into separate pages with a fixed number of items per page. We will also provide the user with options to jump to any page using a select box.

Create a folder for this recipe inside the Chapter8 directory and name it as Recipe5. Using phpMyAdmin, create a table named movies with the following structure:

```
CREATE TABLE IF NOT EXISTS `movies` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `movieName` varchar(64) NOT NULL,
```

```
  PRIMARY KEY (`id`)
);
```

For pagination, we will require a long list so as to enter some movie names in this table, using phpMyAdmin. For this example, we have already inserted 100 names in the table. You can use the movies.sql file that will be supplied along with this book to populate the table. Names of movies in this list have been taken from: http://www.thebest100lists.com/best100movies/.

1. Create a file named index.php inside the Recipe5 folder. In this file, connect to the database and fetch all the movie names from the movies table and create an unordered list with movie names as the list items. Also, create a DIV with ID navigation where the pagination buttons will be placed. Some CSS properties are also defined in the head section for a proper look and feel.

```
<html>
 <head>
<title>Top 100 movies</title>
  <style type="text/css">
  body{ font-family: "Trebuchet MS", Verdana,
      Arial;width:400px;}
  h3{ margin:0;padding:0;}
  ul{ list-style:none;margin:10px 0;padding:0;
    border:1px solid #000;}
  li{ padding:5px;}
  #prev{ float:left;width:100px;}
  #next{ float:right;width:100px;text-align:right;}
  #navigation {float: left; border: 1px solid; padding: 5px;
          width: 97%;}
  #navigation>div { float: left; text-align: center;
            margin-left:40px; 200px;}
  select { width:100px; }
  strong { cursor:pointer; text-decoration:underline;}
  </style>
 </head>
 <body>
  <h3>Top 100 movies voted by people</h3>
  <a href="http://www.thebest100lists.com/best100movies/">
   http://www.thebest100lists.com/best100movies/</a>
  <ul id="list">
<?php
  $mysqli = new mysqli('localhost', 'root', '', 'exampleDB');
  if ($mysqli->connect_errno)
  {
   die('Connect Error: ' . $mysqli->connect_errno);
  }
  $query = 'SELECT movieName FROM movies';

  if ($mysqli->query($query))
  {
   $result = $mysqli->query($query);
   if($result->num_rows > 0)
   {
    while($row = $result->fetch_array())
```

```
      {
        echo '<li>'.$row[0].'</li>';
      }
    }
    else
    {
      echo 'No records';
    }
  }
  else
  {
    echo 'Query Unsuccessful';
  }
?>
  </ul>
  <div id="navigation"></div>
  <p> </p>
 </body>
</html>
```

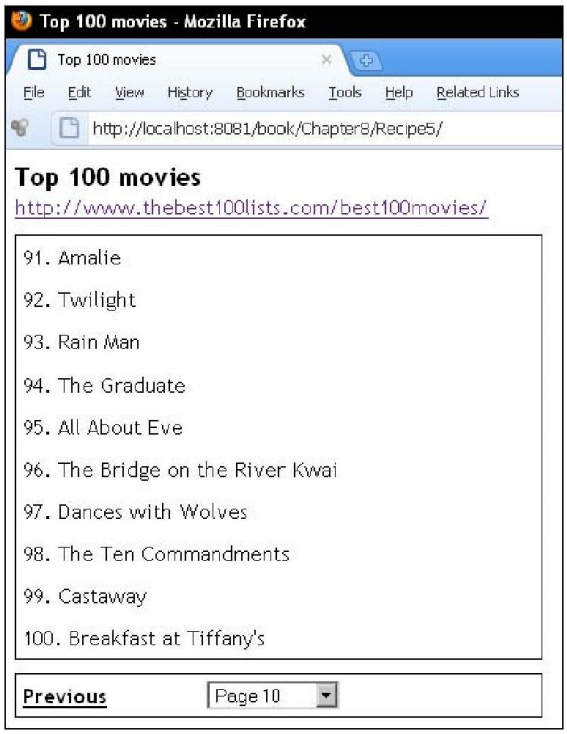
2. The previous screenshot is a partial capture of what the page will look like. It will display all the 100 movies on the browser. Include the jquery.js file and write the jQuery code for paginating this list. First, define the number of items per page and total pages that will be displayed. In this example, we have defined the number of items per page as ten, which means that in total ten pages will be available. Then, define createNavigation function that will create links for the previous page, the next page, and a combo box with all page numbers. Then, write a function setDataAndEvents that will have event handler functions

for these navigation links. Clicking on a navigation link or selecting a page number from the combo box will call another function goToPage that will display the movies for that page only.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  var totalMovies =  $('#list>li').length;
  var moviesPerPage = 10;
  var totalPages = Math.ceil(totalMovies/moviesPerPage);

  createNavigation();
  setDataAndEvents();
  function createNavigation()
  {
   var navHTML = '<strong id="prev">Previous</strong>';
   navHTML+= '<div>';
   navHTML+= '<select id="goTo">';
   navHTML+= '<option value="">Go to page</option>';
   for(var i = 0; i< totalPages; i++)
   {
    navHTML+= '<option value="'+(i+1)+'">Page '+(i+1)+'
     </option>';
   }
   navHTML+= '</select>';
   navHTML+= '</div>';
   navHTML+= '<strong id="next">Next</strong>';

   $('#navigation').html(navHTML);
   $('#prev').hide();
   $('#goTo').val(1);
  }

  function setDataAndEvents()
  {
   $('#list').data('currentPage', 1);
   $('#list>li:gt(' + (moviesPerPage-1) + ')').hide();

   $('#prev').click(function()
   {
    var current = $('#list').data('currentPage');
    goToPage(--current);
   });

   $('#next').click(function(){
    var current = $('#list').data('currentPage');
    goToPage(++current);
   });

   $('#goTo').change(function()
   {
    if($.trim($(this).val()) == '') return;
    goToPage($(this).val());
   });
```

```
  }

  function goToPage(pageNumber)
  {
   if(pageNumber == 1) $('#prev').hide();
       else $('#prev').show();
   if(pageNumber == totalPages) $('#next').hide(); else $('#next').show();

   $('#list').data('currentPage', pageNumber);
   $('#goTo').val(pageNumber);
   var from = (pageNumber – 1) * moviesPerPage;
   var to = from + (moviesPerPage - 1);
   $('#list>li').show();
   $('#list>li:lt(' + (from) + ')').hide();
   $('#list>li:gt(' + (to) + ')').hide();

  }
 });
</script>
```

3. Now run the file in your browser and you will see list of ten movies and navigation links available at the bottom. The page number will be set as 1 in the combo box. Because it will be the first page, only the Next button will be available. Clicking the Next button will change the list as well as page number at the bottom. Going to the last page will hide the Next button.



First of all, we retrieve the list of all movies from the database using query method of mysqli class. Then by iterating over results, we create an unordered list with ID list, using each movie name as a list item. After the list, there is a DIV with ID navigation, which will contain the navigation links. After the page is loaded, jQuery code executes. First,

we get the length of all li element and assign it to the totalMovies variable. Then, we set the moviesPerPage variable to 10. After this, we calculate the total number of pages by
dividing totalMovies with moviesPerPage.

Now, the createNavigation function is called. This function creates two elements inside the navigation DIV that act as Previous and Next buttons and assigns those prev and next IDs respectively. Another select element is created with ID goTo. It has page numbers as the options. Once these elements are created, they are inserted inside the DIV with ID navigation. After that, the Previous button is hidden and the value of select box goTo
is set to 1.

Next is the setDataAndEvents function. To navigate between the previous and next pages, we need to know the current page number and then increase or decrease it for previous or next page respectively. This is achieved by jQuery's data function. We save data with the ul list having currentPage as its key with initial value set to 1. The next line uses the :gt selector that hides all li elements that have an index more than 10 (first page).

Event handlers for Previous and Next buttons come next. On clicking the Previous button, we get the saved value of currentPage; decrease it by 1 and pass it to the goToPage function. Similarly, value of currentPage is increased by 1 for Next button and passed to the function goToPage. The select box has a change event handler attached to it that takes the currently selected value and passes it to the goToPage function.

Function goToPage receives the passed value in the pageNumber variable. Value of this variable is the page where we have to navigate. Here we put two checks. If the user is on the first page, we hide the Previous button, and on last page, we hide the Next button. Then, we update the value of currentPage and then set the value of select box to pageNumber. To decide what list items are to be displayed for that page, we calculate two variables: from and to. The final three lines hide all other list items except the ones which do not fall in range between from and to.

Perhaps the simplest example of explaining auto-suggest is the Google homepage. When
you type a query in the search box, it displays a list of queries beneath it by matching your search terms.

We will create an example with the same functionality where text entered by the user will be matched against user names in a table and matching results will be displayed to the user in the form of a list just below the textbox in form of suggestions. The user will be able to use arrow keys to navigate up or down in a list and select a name from the list.

Create a folder named Recipe6 inside the Chapter8 directory. To be able to match user input with the database, we will require a table. Open phpMyAdmin and create a new table named users with the following structure and data:

```
CREATE TABLE `users` (
 `id` int(11) NOT NULL AUTO_INCREMENT,
 `username` varchar(32) NOT NULL,
 `password` varchar(32) NOT NULL,
 PRIMARY KEY (`id`)
```

```
);


INSERT INTO `users` (`id`, `username`, `password`) VALUES
(1, 'holmes', 'sherlockholmes'),
(2, 'watson', 'johnwatson'),
(3, 'sati', 'pranay'),
(4, 'mantu', 'ajayjoshi'),
(5, 'sahji', 'brijsah'),
(6, 'vijay', 'vijayjoshi'),
(7, 'brij', 'brijsah'),
(8, 'arjun', 'samant'),
(9, 'jyotsna', 'sonawane'),
(12, 'ravindra', 'pokharia'),
(13, 'prakash', 'joshi'),
(14, 'sahji2', 'aloklal'),
(15, 'basant', 'bhandari'),


(16, 'ajay', 'gamer')
```

1. Create a file named index.html inside the Recipe6 folder. In this file, create a DIV with class autosuggest. Inside this DIV, create a textbox with ID suggest, and an unordered list with ID suggestions. This list will display the matched results. Now, create an image tag that will have a spinning loading indicator that will be displayed while script is busy getting data from the database. Finally, create a span element with ID error that will be displayed when there are no matched results.

```
<html>
 <head>
  <title>Autocomplete</title>
  <link rel="stylesheet" type="text/css" href="style.css">
 </head>
 <body>
  <div class="autosuggest">
   <input type="text" id="suggest"/>
   <ul id="suggestions">
   </ul>
   <img src="ajax-loader.gif" alt="loading" title="loading"
    id="loader"/>
   <span id="error"></span>
  </div>
 </body>
</html>
```

2. Note that we have referred to a style.css file in the head section. CSS attributes are very important for this example as we have to position the ul, just under the textbox. Create a new file named style.css and place the following CSS properties in it:

```
body{ font-family: "Trebuchet MS", verdana, arial;width:400px;margin:0 auto; }
.autosuggest
{
```

```
 width:200px;
 top:5px;
 position:relative;
}
input { width:200px;}
#suggestions
{
 position:absolute;
 list-style:none;
 margin:0;
 padding:0;
 width:200px;
 display:none;
 background-color:#ECECF6;
 top:20px;
 left:0px;
}
#suggestions li
{
 cursor:pointer;
 padding:5px;
 border-right:1px solid #000;
 border-bottom:1px solid #000;
 border-left:1px solid #000;

}
.active
{
 background-color:red;
 color:#fff;
}
#error
{
 top:25px;
 font-weight:bold;
 color:#ff0000;
}
#loader
{
 position:absolute;
 top:2px;
 right:0;
 display:none;
}
```

3. Focusing on jQuery now, add the jquery.js file before the closing of the body tag. Now define four event handlers that will get the suggestions from the database and display them in a list at a proper position. Call function getSuggestions on keyup. This is the core function that picks up keystrokes and gets matching results using an AJAX request. Value of textbox is sent through an AJAX request to a PHP file, suggestions.php. On receiving the results function, showSuggestions executes, which creates a list from received data and displays it.
4. Function navigateList will be executed on keydown event. It will take care of the navigation by adding functionality for up and down arrow keys and the *Enter* key for selecting a list item. Next are two

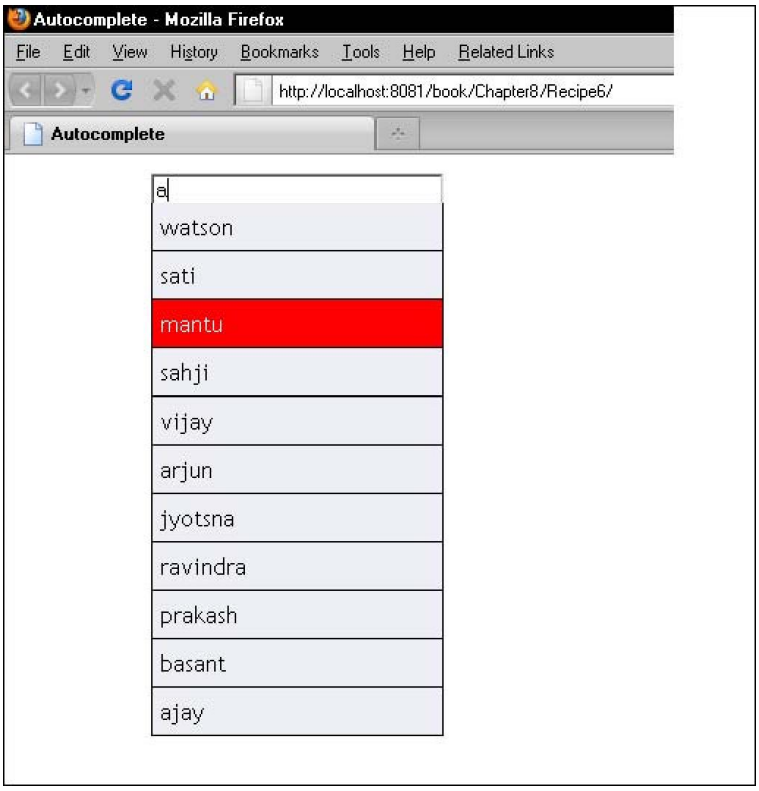
functions for mouse movements. The first function listHover will execute whenever the mouse pointer enters or leaves a list item and will change the look and feel of list items. listClick function will be used to fill the textbox with the selected value when a mouse is clicked against a list item.

```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  var xhr;
  $('#suggest').keyup(getSuggestions);
  $('#suggest').keydown(navigateList);
  $('#suggestions>li').live(0'mouseover mouseout click',
   listHover);

  function getSuggestions(event)
  {
   var value = jQuery.trim($(this).val());
   if(value == '' || event.which == 27)
   {
    $('#suggestions').empty().hide();
    $('#loader').hide();
   }

   if((event.which >= 65 && event.which <= 90) ||
    event.which == 8 || event.which == 46)
   {
    $('#loader').show();
    if(xhr) xhr.abort();
    if(value.length >= 1)
    {
     xhr = $.getJSON
      (
       'suggestions.php',
       { input : value },
       showSuggestions
      );
    }
    else
    {
     $('#loader').hide();
    }
   }
  }

  function showSuggestions(data)
  {
   if(data == false)
   {
    $('#error').html('No results').show();
    $('#suggestions').empty().hide();
   }
   else
   {
    var str = '';
```

```
   $('#error').empty().hide();
   for(var i=0; i < data.length; i++)
   {
    str+= '<li>'+data[i]+'</li>';
   }
   $('#suggestions').html(str).show();
  }
  $('#loader').hide();
}


function navigateList(event)
{
  switch(event.which)
  {
    case 38:  //up arrow
     if($('#suggestions>li.active').length > 0)
     {
       $('#suggestions>li.active').removeClass('active').
         prev().addClass('active');
     }
     else
     {
       $('#suggestions>li:last').addClass('active');
     }
    break;

    case 40:  //down arrow
     if($('#suggestions>li.active').length > 0)
     {
       $('#suggestions>li.active').removeClass('active').
         next().addClass('active');
     }
     else
     {
       $('#suggestions>li:first').addClass('active');
     }
    break;

    case 13:          //enter
           $('#suggest').val($('#suggestions>li.active').
          html());
     $('#suggestions').empty().hide();
    break;
  }
}

function listHover(event)
{
  if (event.type == 'mouseover')
  {
    $('#suggestions>li.active').removeClass('active');
  }
  $(this).toggleClass('active');
```

```
    if(event.type == 'click')
    {
      $('#suggest').val($(this).html());
      $(this).parent().empty().hide();
      $('#suggest').focus();
    }
   }
 });
</script>
```

5. Create another file named suggestions.php in the same directory. Connect to the exampleDB database in this file, and using the value of textbox, write a query to fetch results from the database. Once results are retrieved, JSON is created and is sent back to the browser where it is displayed by jQuery.
6. Run the index.html file in the browser and press any key. The AJAX request will try to get the matching results and will show them in the list. Below is a sample response after pressing key a:



First, we will make sure that the ul always appears below the textbox. There is a clean and easy way to do it. First, make the CSS position of the outer DIV relative. This has been done in the CSS file. Now you can make the position of any element inside this DIV absolute, relative to the DIV. So, the following CSS properties of ul will place it just below the textbox.

```
position:absolute;
top:20px;
left:0px;
```

Rest of the properties define the look and feel for the ul. Similarly, we place the loaded image absolutely to the right.

Let us implement autocomplete now. First is the keyup event handler for the textbox. It executes a function getSuggestions. This function gets the value of the textbox and continues only if the value is not empty. Then, it checks which keys are pressed using
event.which that is provided by jQuery. Pressing keys between *a-z*, *A-Z*, *Delete* key, or the *Backspace* key will change the value of textbox. So, we take this value from the textbox and send it with an AJAX request to the suggestions.php file. A callback function showSuggestions is provided for handling the response. suggestions.php returns a JSON that is used in showSuggestions. The response can be an array of matching names or false in case of any error or upon finding no records. If the response from showSuggestions is false, we show an error message. Otherwise we iterate over the response array and create a list item for each element in the array. After all list items are created, we insert them into the ul with ID suggestions. Just before the request is sent, we show the loading indicator image and after processing is done in showSuggestions we hide it.

We want to be able to use arrow keys to move up or down in the list and select a value by pressing *Enter*. Moving up and down in the list will highlight the item by adding a CSS class active to it. For this purpose, another event handler navigateList has been defined for the keyDown event. This function has a switch statement with three cases. First one is for Up arrow key whose key code is 38. It checks if any li element has already CSS class active or not. If not, it adds the active class to the last element that highlights the last item in the list. If a list item already has an active class attached to it, then on pressing the Up arrow key, an if condition is executed that removes the active class from the highlighted element and adds the same class to its previous element.

The code for the Down arrow key works in a similar way. If no element is highlighted and the Down arrow key is pressed, the first element of the list elements is selected If an item is already active and Down arrow key is pressed again, the active class is removed from it and is added to the next element.

The third and final case is for the *Enter* key, which has key code 13. On pressing *Enter*, the HTML of the currently-highlighted element is taken and is set as the value of the textbox.
After that, the ul suggestions are emptied and hidden.

After keyboard navigation, we need to take care of mouse selections too. Hovering over a list item should add an active class to it and moving the mouse pointer out of it should remove this class. Also, clicking an item should select its value in the textbox. As no list item is present inside the ul tag at the beginning, we use the live method to add the listHover event handler. This function will execute whenever the mouse pointer enters a list item, leaves it, or it is clicked. In this function, if the event is mouseover, we first remove the active class from any previously active item. Then we use the toggleClass function to add or remove the active class from the current item. This will make a list item active when mouse pointer is over it and will remove the active class when the mouse pointer is taken away.

Finally, listHover also checks if a li was clicked, we take the active item's HTML and insert it into the textbox. Then the ul is emptied and hidden and focus is given to textbox.

On the server side, the PHP file suggestions.php receives the value of the textbox and queries the users table in the database to find all the matching records.

```
$query = 'SELECT username FROM users where username like "%'.
 $_GET['input']. '%"';
```

Use of % before and after the textbox value in our query indicates that any characters may precede or follow the value. This means if the input value was "ss", it will match both "pass" and "passed". After getting the results from the database, we iterate over them and create an array. This array is converted to JSON and echoed back to the browser.

Another important thing to note is variable xhr, which we have declared at the beginning of the file. If the user presses multiple keys, that number of requests will hit the server simultaneously. To avoid this, we assign

$.getJSON to variable xhr. Now before sending a request to the server, we can abort any previous request using the abort method of xmlHttpRequest so that only the current request is processed.

Creating keyboard shortcuts in Chapter 1

A tag cloud is a visual representation of tags or keywords where each tag's size or color is determined by its weight. Consider a blog with many articles. Each article can be tagged to a category like PHP, jQuery, XML, JSON, and so on. Out of these, if PHP category has 50 articles, jQuery has 30, XML 10, and JSON has 22 articles, we can say that PHP has most weight and XML has the least weight. If we wanted to present these tags in a graphical manner so that a more weighted item is more emphasized, we can do so by setting their respective font size in proportion to their weights.

We will create a similar example where we have a list of cities in a database and each has a rating out of 100. We will present these tags in the form of a tag cloud such as with their sizes depending on their rating.

Create a folder named Recipe7 inside the Chapter8 directory. For the list of cities and their ratings, use the following SQL query in phpMyAdmin to create a new table named cities:

```
CREATE TABLE `cities` (
  `id` int(3) NOT NULL AUTO_INCREMENT,
  `cityName` varchar(32) NOT NULL,
  `cityRating` int(3) NOT NULL,
  PRIMARY KEY (`id`)
);

INSERT INTO `cities` (`id`, `cityName`, `cityRating`) VALUES
(1, 'Udaipur', 71),
(2, 'Leh', 55),
(3, 'Mahabaleshwar', 28),
(4, 'Mount Abu', 31),
(5, 'Rishikesh', 15),
(6, 'Hampi', 81),
(7, 'Matheran', 29),
(8, 'Manali', 85),
(9, 'Mysore', 33),
(10, 'Jaipur', 55),
(11, 'Munnar', 89),
(12, 'Bangalore', 66),
(13, 'Wayanad', 42),
(14, 'Amritsar', 29),
(15, 'Gangtok', 69),
(16, 'Havelock Islands', 27),
(17, 'DharamShala', 57),
(18, 'Kashmir', 78),
(19, 'Tirupati', 22),


(20, 'Goa', 75)
```

1. Create a file named index.html in the Recipe7 folder. In this file, create a DIV with cloud ID and define some CSS styles for DIV and anchor elements that will be created in the page.

```
<html>
 <head>
  <title>Create a tag cloud</title>
  <style type="text/css">
  body { font-family:"Trebuchet MS",Verdana,Arial; }
  div
  {
   width:600px;
   border:1px solid;
   float:left;
   position:relative;
  }
  a
  {
   float:left;
   text-decoration:none;
   padding:0px 5px;
   text-transform:lowercase;
  }
  span { font-size:12px; }
  </style>
 </head>
 <body>
  <h3>Popularity of Indian Tourist Destinations</h3>
  <div id="cloud"></div>
 </body>
</html>
```

2. Include the jquery.js file before closing the body tag. In jQuery code, send an AJAX request to the PHP file tags.php. Callback function is createTagCloud for this AJAX call. This function iterates over the response and creates tags on the page.

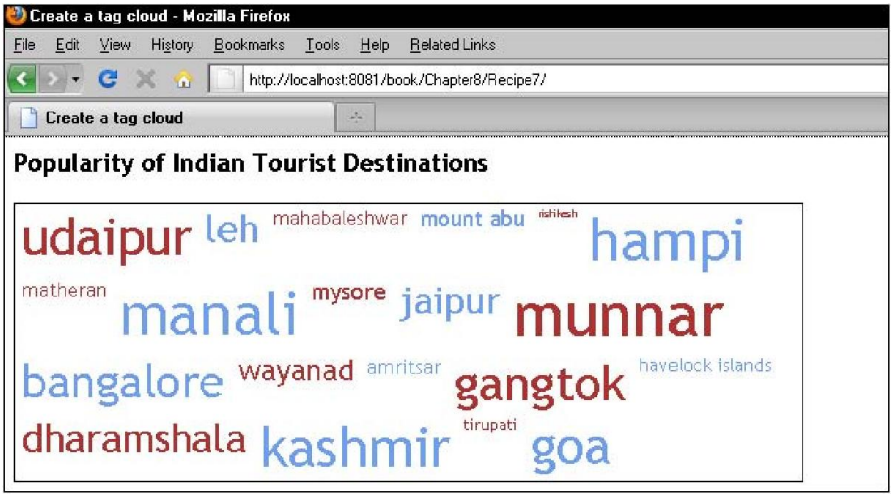
```
<script type="text/javascript" src="../jquery.js"></script>
<script type="text/javascript">
 $(document).ready(function()
 {
  $.getJSON(
   'tags.php',
   {},
   createTagCloud
  );
 });

 function createTagCloud(response)
 {
  var str = '', i=0;
  $.each(response.tags, function(index, tag)
  {
   var color = i%2 == 0 ? 'color:#A52A2A' : 'color:#6495ED';
   var fontSize = ((parseInt(tag.rating,10)/30));
```

```
    str+= '<a href="#" style="font-size:'+fontSize+'em;'+color+'
      " title="' + tag.city + '">' + tag.city + '</a>';
    i++;
   });
   $('#cloud').html(str);
 }
</script>
```

3. Create another file named tags.php. This file will connect to the database and will fetch the city information from the cities table. A JSON string will be created from the database results that will be sent to the browser where jQuery receives it and handles the tag creation.

```
<?php
$mysqli = new mysqli('localhost', 'root', '', 'exampleDB');

if (mysqli_connect_errno())
{
   die('Unable to connect!');
}
$query = 'SELECT cityName, cityRating FROM cities';
$arr = array();
if ($result = $mysqli->query($query))
{
   if ($result->num_rows > 0)
 {
   while($row = $result->fetch_assoc())
   {
     array_push($arr, array('city' => $row['cityName'],
         'rating' => $row['cityRating']));
   }
 }
}
$result = array('tags' => $arr);
header('Content-Type:text/json');
echo json_encode($result);
?>
```

4. Run the index.php file in the browser and you will see a collection of city names in various sizes.



Once the document is ready, an AJAX request is sent to the PHP file tags.php using $.getJSON method. The callback function for this request is createTagCloud. In the
tags.php file, a SELECT query is executed which fetches all city names and their ratings. Then we use the fetch_assoc method to retrieve results from each row and insert them
into the $arr array.

Once all records are pushed in this array $arr, we assign it to an associative array $result having tags as the key.

Finally, we set the response type as text/json and convert the array $result to a JSON string using PHP's json_encode method. The JSON will look like the following:


```
{
    "tags":
    [
        {"city":"Udaipur","rating":"71"},
        {"city":"Leh","rating":"55"},
        {"city":"Mahabaleshwar","rating":"28"},
        {"city":"Mount Abu","rating":"31"},
        {"city":"Rishikesh","rating":"15"},
        {"city":"Hampi","rating":"81"},
        {"city":"Matheran","rating":"29"},
        {"city":"Manali","rating":"85"},
        {"city":"Mysore","rating":"33"},
        {"city":"Jaipur","rating":"55"},
        {"city":"Munnar","rating":"89"},
        {"city":"Bangalore","rating":"66"},
        {"city":"Wayanad","rating":"42"},
        {"city":"Amritsar","rating":"29"},
        {"city":"Gangtok","rating":"69"},
        {"city":"Havelock Islands","rating":"27"},
        {"city":"DharamShala","rating":"57"},
        {"city":"Kashmir","rating":"78"},
        {"city":"Tirupati","rating":"22"},
        {"city":"Goa","rating":"75"}
    ]
}
```


Now the response is available in the createTagCloud function inside a variable named response. We use jQuery's each method to iterate over the tags array in this JSON. For each element, we set different colors for alternate tags by checking the value of variable i. For deciding the font size, we divide the rating by 30. You can choose any number for division, depending on how large or small the font sizes need to be. Once the font size and colors are set, we create anchor tags, set these values, and keep on appending these anchors to a variable str. After the array has been traversed fully, we insert the value of variable str into DIV with ID cloud. The end result is a beautiful tag cloud.

7

jQuery UI widgets

This chapter covers

. Extending the set of HTML controls with jQuery UI widgets
. Augmenting HTML buttons
. Using slider and datepicker controls for numeric and date input
. Showing progress visually
. Simplifying long lists with autocompleters
. Organizing content with tabs and accordions
. Creating dialog boxes

Since the dawn of the web, developers have been constrained by the limited set of controls afforded by HTML. Although that set of controls runs the gamut from simple text entry through complex file selection, the variety of provided controls pales in comparison to those available to desktop application developers. HTML 5 promises to expand this set of controls, but it may be some time before support appears in all major browsers.

For example, how often have you heard the HTML <select> element referred to as a "combo box," a desktop control to which it bears only a passing resem blance? The real combo box is a very useful control that appears often in desktop applications, yet web developers have been denied its advantages.

But as computers have become more powerful, browsers have increased their capabilities, and DOM manipulation has become a commonplace activity, clever web developers have been taking up the slack. By creating extended controls—either augmenting the existing HTML controls or creating controls from scratch using basic elements—the developer community has shown nothing short of sheer ingenuity in using the tools at hand to make the proverbial lemonade from lemons.

Standing on the shoulders of core jQuery, jQuery UI brings this ingenuity to us, as jQuery users, by providing a set of custom controls to solve common input problems that have traditionally been difficult to solve using the basic control set. Be it making standard elements play well (and look good) in concert with other elements, accepting numeric values within a range, allowing the specification of date values, or giving us new ways to organize our content, jQuery UI offers a valuable set of widgets that we can use on our pages to make data entry a much more pleasurable experience for our users (all while making it easier on us as well).

Following our discussion of the core interactions provided by jQuery UI, we'll continue our exploration by seeing how jQuery UI fills in some gaps that the HTML control set leaves by providing custom controls (widgets) that give us more options for accepting user input. In this chapter, we'll explore the following jQuery UI widgets:

. Buttons (section 11.1)
. Sliders (section 11.2)
. Progress bars (section 11.3)
. Autocompleters (section 11.4)
. Datepickers (section 11.5)
. Tabs (section 11.6)
. Accordions (section 11.7)
. Dialog boxes (section 11.8)

Like the previous chapter, this is a long one! And as with interactions, the jQuery UI methods that create widgets follow a distinct pattern that makes them easy to understand. But unlike interactions, the widgets pretty much stand on their own, so you can choose to skip around the sections in this chapter in any order you like.

We'll start with one of the simpler widgets that lets us modify the style of existing control elements: buttons.

Buttons and buttonsets

At the same time that we lament the lack of variety in the set of HTML 4 controls, it offers a great number of button controls, many of which overlap in function.

There's the <button> element, and no less than six varieties of the <input> element that sport button semantics: button, submit, reset, image, checkbox, and radio. Moreover, the <button> element has subtypes of button, submit, and reset, whose semantics overlap those of the corresponding input element types.

NOTE Why are there so many HTML button types? Originally, only the <input> button types were available, but as they could only be defined with a simple text string, they were deemed limiting. The <button> element was added later; it can contain other elements and thereby offers more rendering possibilities. The simpler <input> varieties were never deprecated, so we've ended up with the plethora of overlapping button types.

All these buttons types offer varying semantics, and they're very useful within our pages. But, as we'll see when we explore more of the jQuery UI widget set, their default visual style may not blend well with the styles that the various widgets exhibit.

Button appearance within UI themes

Remember back when we downloaded jQuery UI near the beginning of chapter 9? We were given a choice of various themes to download, each of which applies a different look to the jQuery UI elements.

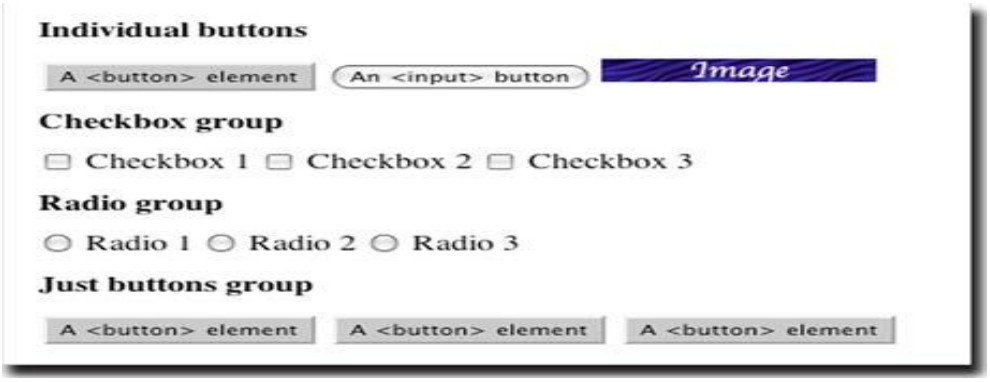
To make our buttons match these styles, we *could* poke around the CSS file for the chosen theme and try to find styles that we could apply to the button elements to bring them more into line with how the other elements look. But as it turns out, we don't have to—jQuery UI provides a means to augment our button controls so their appearance matches the theme without changing the semantics of the elements. Moreover, it will also give them hover styles that will change their appearance slightly when the mouse pointer hovers over them—something the unstyled buttons lack.

The button() method will modify individual buttons to augment their appearance, while the

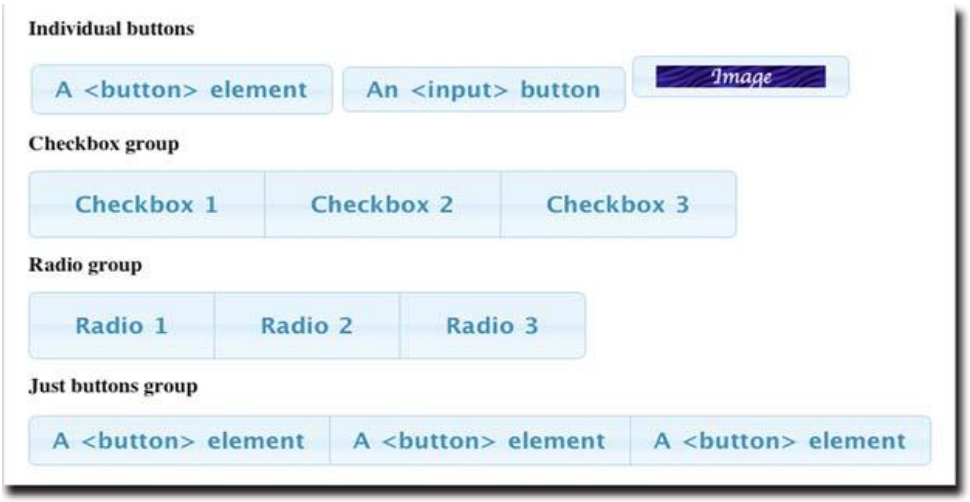
buttonset() method will act upon a set of buttons (most often a set of radio buttons or checkboxes) not only to theme them, but to make them appear as a cohesive unit.

Consider the display in figure 11.1.

This page fragment shows the unthemed display of some individual button elements, and some groupings of checkboxes, radio buttons, and <button> elements. All perfectly functional, but not exactly visually exciting.



Buttons and buttonsets



After applying the button() method to the individual buttons, and the buttonset() method to the button groups (in a page using the Cupertino theme), the display changes to that shown in figure 11.2.

After styling, the new buttons make those shown in figure 11.1 look positively Spartan.

Not only has the appearance of the buttons been altered to match the theme, the groups have been styled so that the buttons in the group form a visual unit to match their logical grouping. And even though the radio buttons and checkboxes have been restyled to look like “normal” buttons, they still retain their semantic behaviors. We’ll see that in action when we introduce the jQuery UI Buttons Lab.

This theme’s styling is one we’ll become very familiar with as we progress through the jQuery UI widgets in the remainder of this chapter. But first we’ll take a look at the methods that apply this styling to the button elements.

Creating themed buttons

The methods that jQuery UI provides to create widgets follow the same style we saw in the previous chapter for the interaction methods: calling the button() method and passing an options hash creates the widget in the first place, and calling the same method again but passing a string that identifies a widget-targeted operation modifies the widget.

The syntax for the button() and buttonset() methods is similar to the methods we investigated for the UI interactions:

Command syntax: button and buttonset

button('disable')
button('enable')
button('destroy')
button('option',optionName,value)
buttonset(options)
buttonset('disable')
buttonset('enable')
buttonset('destroy')
buttonset('option',optionName,value)

Themes the elements in the wrapped set to match the currently loaded jQuery UI theme. Button appearance and semantics will be applied even to non-button element such as and
t('enable')buttonset('destroy')buttonset('option',optionName,value)
Themes the elements in the wrapped set to match the currently loaded jQuery UI theme. Button appearance and semantics will be applied even to non-button element such as and

.

Parameters

options	(Object) An object hash of the options to be applied to the elements in the wrapped set, as described in table 11.1, making them themed buttons.
'disable'	(String) Disables click events for the elements in the wrapped set.
'enable'	(String) Re-enables button semantics for the elements in the wrapped set.
'destroy'	(String) Reverts the elements to their original state, before applying the UI theme.
'option'	(String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a jQuery UI button element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.
optionName	(String) The name of the option (see table 11.1) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.

value (Object) The value to be set for the option identified by the optionName parameter.

Returns

The wrapped set, except for the case where an option value is returned.

To apply button *theming* to a set of elements, we call the button() or buttonset() method with a set of options, or with no parameter to accept the default options. Here's an example:

```
$(':button').button({ text: true });
```

NOTE The word "theming" isn't in the dictionary, but it's what jQuery UI uses, so we're running with it.

The options that are available to use when creating buttons are shown in table 11.1.

The button() method comes with plenty of options, and you can try them out in the Buttons Lab page, which you'll find in chapter11/buttons/lab.buttons.html and is shown in figure 11.3.

Follow along in this Lab as you read through the options list in table 11.1.

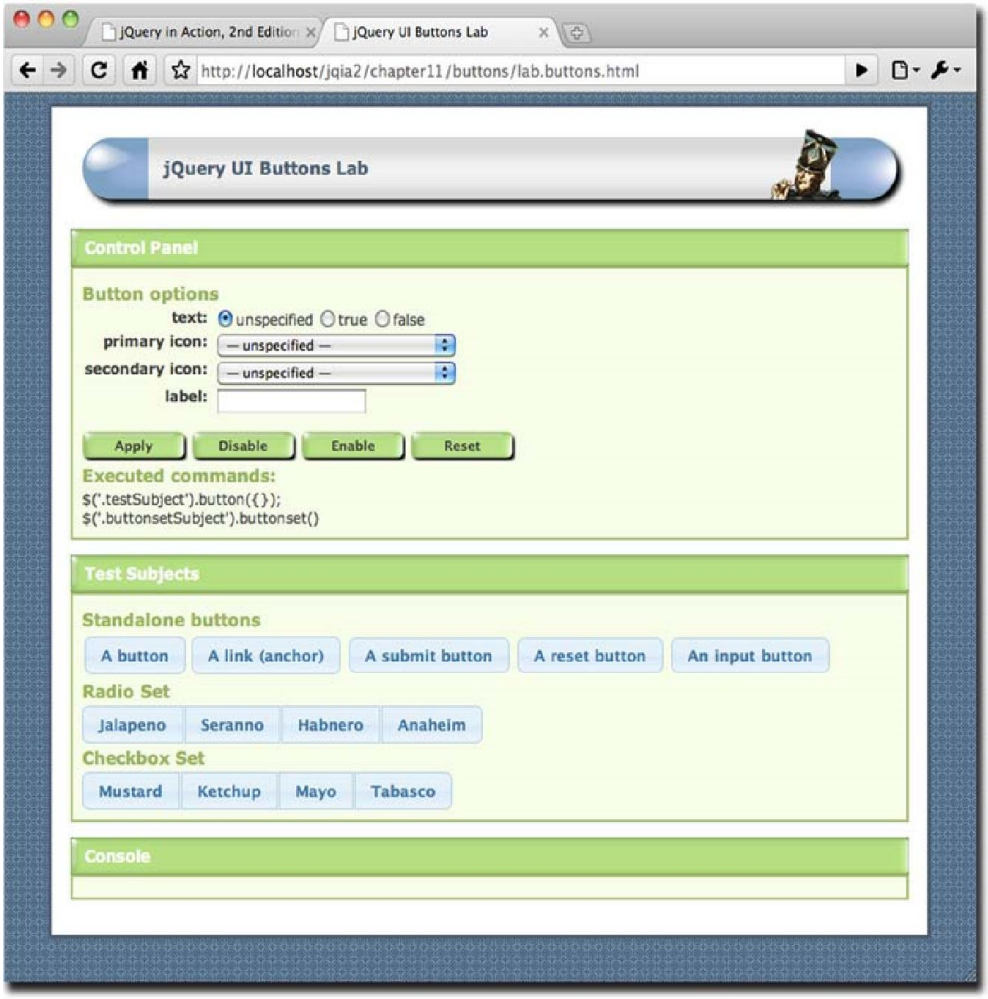


Table 11.1 Options for the jQuery UI buttons and buttonsets

Option	Description	In Lab?
icons	(Object) Specifies that one or two icons are to be displayed in the button: primary icons to the left, secondary icons to the right. The primary icon is identified by the primary property of the object, and the secondary icon is identified by the secondary property.The values for these properties must be one of the 174 supported call names that correspond to the jQuery button icon set. We'll discuss these in a moment.If omitted, no icons are displayed.	✓

Table 11.1 Options for the jQuery UI buttons and buttonsets *(continued)*

Option	Description	In Lab?
label (String)	Specifies text to display on the button that overrides the natural label. If omitted, the natural label for the element is displayed. In the case of radio buttons and checkboxes, the natural label is the <label> element associated with the control.	✓
text (Boolean)	Specifies whether text is to be displayed on the button. If specified as false, text is suppressed if (and only if) the icons option specifies at least one icon. By default, text is displayed.	✓

These options are straightforward except for the icons options. Let's chat a little about that.

Button icons

jQuery UI supplies a set of 174 themed icons that can be displayed on buttons. You can show a single icon on the left (the primary icon), or one on the left and one on the right (as a secondary icon).

Icons are specified as a class name that identifies the icon. For example, to create a button with an icon that represents a little wrench, we'd use this code:

```
$('#wrenchButton').button({  icons: { primary: 'ui-icon-wrench' }
});
```

If we wanted a star on the left, and a heart on the right, we'd do this:

```
$('#weirdButton').button({  icons: { primary: 'ui-icon-star', secondary: 'ui-icon-heart' }
});
```

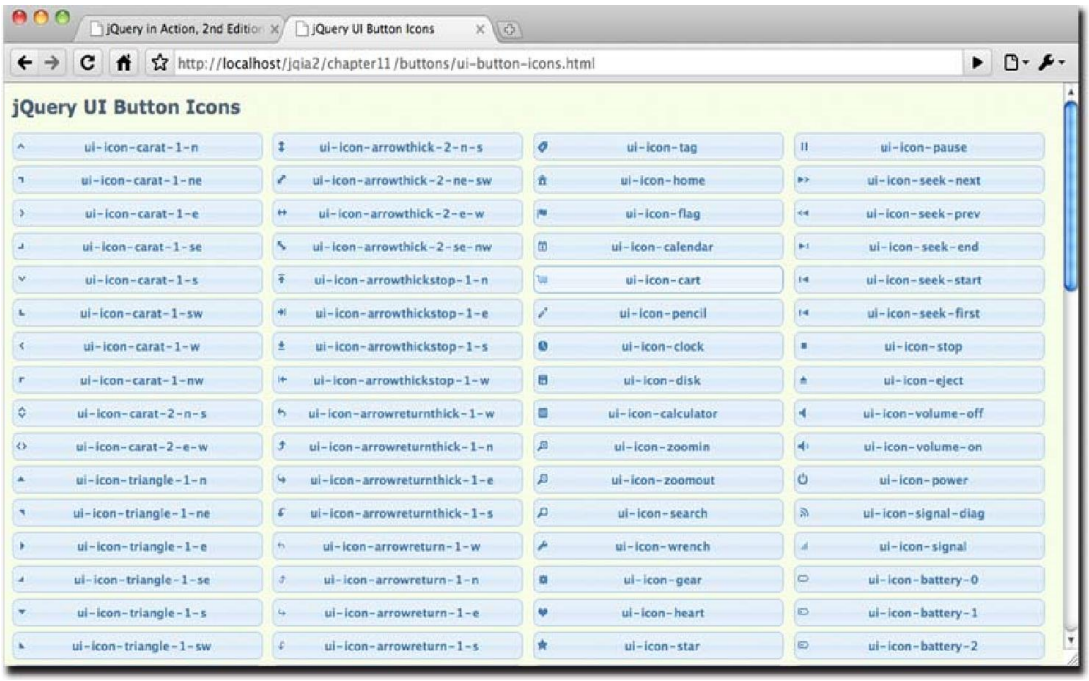
Because we all know how many words a picture is worth, rather than just listing the available icon names here, we've provided a page that creates a button for each of the icons, labeled with the name of the icon. You'll find this page at chapter11/buttons/ ui-button-icons.html, and it's shown in figure 11.4.

You might want to keep this page handy for whenever you want to find an icon to use on your buttons.

Button events

Unlike the interactions and the remainder of the widgets, there are no custom events associated with jQuery UI buttons.

Because these widgets are merely themed versions of existing HTML 4 controls, the native events can be used as if the buttons had not been augmented. To handle button clicks, we simply continue to handle click events for the buttons.



Styling buttons

The whole purpose of using the jQuery UI button() and buttonset() methods is to make the buttons match the chosen jQuery UI theme. It's as if we took the buttons and sent them to appear on *What Not to Wear* (a popular US and UK TV makeover reality show); they start out drab and homely and emerge looking fabulous! But even so, we may want to fine-tune those styled elements to make them work better on our pages. For example, the text of the buttons on the Buttons Icon page was made smaller to fit the buttons on the page.

jQuery UI augments or creates new elements when creating widgets, and it applies class names to the elements that match the style rules in the theme's CSS style sheet. We can use these class names ourselves to augment or override the theme definitions on our pages.

For example, in the Button Icons page, the button text's font size was adjusted like this:

```
.ui-button-text { font-size: 0.8em;}
```

The class name ui-button-text is applied to the element that contains the button text.

It would be nearly impossible to cover all the permutations of elements, options, and class names for the widgets created by jQuery UI, so we're not even going to try. Rather, the approach that we'll take is to provide, for each widget type, some tips on some of the most common styling that we're likely to need on our pages. The previous tip on restyling the button text is a good example.

Button controls are great for initiating actions, but except for radio buttons and checkboxes, they don't represent values that we might want to obtain from the user. A number of the jQuery UI widgets represent logical form controls that make it easy for us to obtain input types that have long been an exercise in pain. Let's take a look at one that eases the burden of obtaining numeric input.

Sliders

Numeric input has traditionally been a thorn in the side of web developers everywhere. The HTML 4 control set just doesn't have a control that's well suited to accepting numeric input.

A text field can be (and is most often) used to accept numeric input. This is less than optimal because the value must be converted and validated to make sure that the user doesn't enter "xyz" for their age or for the number of years they've been at their residence.

Although after-the-fact validation isn't the greatest of user experiences, filtering the input to the text control such that only digits can be entered has its own issues. Users might be confused when they keep hitting the *A* key and nothing happens.

In desktop applications, a control called a *slider* is often used whenever a numeric value within a certain range is to be obtained. The advantage of a slider over text input is that it becomes impossible for the user to enter a bad value. Any value that they can pick with the slider is valid.

jQuery UI brings us a slider control so that we can share that advantage.

Creating slider widgets

A slider generally takes the form of a "trough" that contains a handle. The handle can be moved along the trough to indicate the value selected within the range, or the user can click within the trough to indicate where the handle should move to within the range.

Sliders can be arranged either horizontally or vertically. Figure 11.5 shows an example of a horizontal slider from a desktop application.

Unlike the button() method, sliders aren't created by augmenting an existing HTML control. Rather, they're composited from basic elements like and <a>. The target element is styled to form the trough of the slider, and anchor elements are created within it to form the handles.

The slider widget can possess any number of handles and can therefore represent any number of values. Values are specified using an array, with one entry for each handle. However, as the single-handle case is so much more common than the multi-handle case, there are methods and options that *treat* the slider as if it had a single value.

This prevents us from having to deal with arrays of a single element for the way that we'll use sliders most often. Thanks jQuery UI team! We appreciate it!

This is the method syntax for the slider() method:

Command syntax: slider

slider(options)
slider('disable')
slider('enable')
slider('destroy')
slider('option',optionName,value)
slider('value',value)
slider('values',index,values)

Transforms the target elements (elements recommended) into a slider control.

Parameters

options (Object) An object hash of the options to be applied to the elements in the wrapped set, as described in table 11.2 making them sliders.

'disable' (String) Disables slider controls.

'enable' (String) Re-enables disabled slider controls.

'destroy' (String) Reverts any elements transformed into slider controls to their previous state.

'option'(String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a slider element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.

optionName(String) The name of the option (see table 11.2) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.

value(Object) The value to be set for the option identified by the optionName parameter (when used with 'option'), the value to be set for the slider (if used with 'value'), or the value to be set for the handles (if used with 'values').

'value'(String) If value is provided, sets that value for the single-handle slider and returns that value ; otherwise the slider's current value is returned.

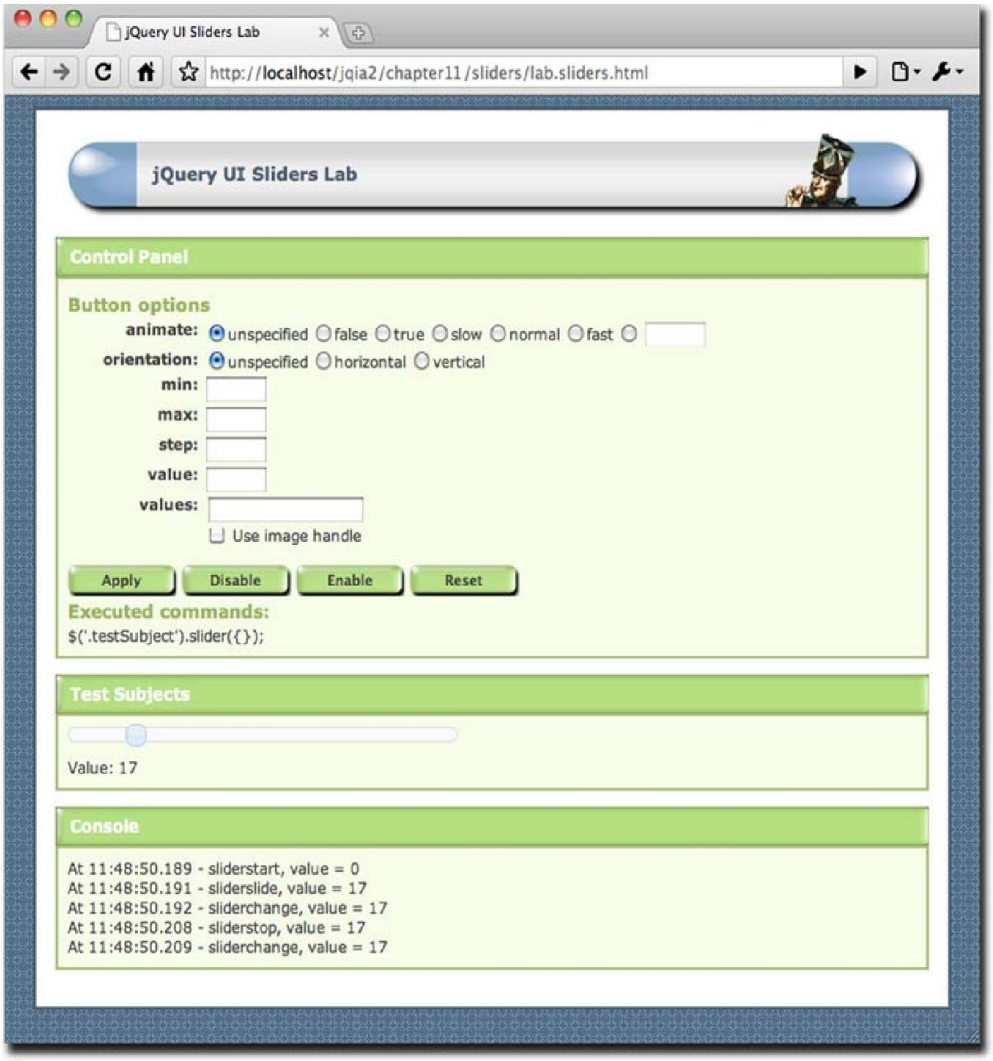
'values'(String) For sliders with multiple handles, gets or sets the value for specific handles, where the index parameter must be specified to identify the handles. If the values parameter is provided, sets the value for the handles. The values of the specified handles are returned.

index(Number|Array) The index, or array of indexes, of the handles to which newvalues are to be assigned.

Returns

The wrapped set, except for the case where an option or handle value is returned.

When creating a slider, there are a good variety of options for creating slider controls with various behaviors and appearance.



 While you're reading through the options in table 11.2, bring up the Sliders Lab in file chapter11/sliders/lab.sliders.html (shown in figure 11.6), and follow along, trying out the various options. Now let's explore the events that slider controls can trigger.

Table 11.2 Options for the jQuery UI sliders

Option	Description	In Lab?
`animate`	(Boolean|String|Number) If `true`, causes the handle to move smoothly to a position clicked to within the trough. Can also be a duration value or one of the strings `slow`, `normal`, or `fast`. By default, the handle is moved instantaneously.	✓
`change`	(Function) Specifies a function to be established on the slider as an event handler for slidechange events. See the description of the slider events in table 11.3 for details on the information passed to this handler.	✓
`max`	(Number) Specifies the upper value of the range that the slider can attain—the value represented when the handle is moved to the far right (for horizontal sliders) or top (for vertical sliders). By default, the maximum value of the range is 100.	✓
`min`	(Number) Specifies the lower value of the range that the slider can attain—the value represented when the handle is moved to the far left (for horizontal sliders) or bottom (for vertical sliders). By default, the minimum value of the range is 0.	✓
`orientation`	(String) One of `horizontal` or `vertical`. Defaults to `horizontal`.	✓
`range`	(Boolean|String) If specified as `true`, and the slider has exactly two handles, an element that can be styled is created between the handles. If the slider has a single handle, specifying `min` or `max` creates a range element from the handle to the beginning or end of the slider respectively. By default, no range element is created.	✓
`start`	(Function) Specifies a function to be established on the sliders as an event handler for slidestart events. See the description of the slider events in table 11.3 for details on the information passed to this handler.	✓
`slide`	(Function) Specifies a function to be established on the slider as an event handler for slide events. See the description of the slider events in table 11.3 for details on the information passed to this handler.	✓
`step`	(Number) Specifies discrete intervals between the minimum and maximum values that the slider is allowed to represent. For example, a step value of 2 would allow only even numbers to be selected. The step value should evenly divide the range. By default, `step` is `1` so that all values can be selected.	✓
`stop`	(Function) Specifies a function to be established on the slider as an event handler for slidestop events. See the description of the slider events in table 11.3 for details on the information passed to this handler.	✓
`value`	(Number) Specifies the initial value of a single-handle slider. If there are multiple handles (see the `values` options), specifies the value for the first handle. If omitted, the initial value is the `minimum` value of the slider.	✓

Option	Description	In Lab?
values	(Array) Causes multiple handles to be created and specifies the initial values for those handles. This option should be an array of possible values, one for each handle.For example, [10,20,30] will cause the slider to have three handles with initial values of 10, 20, and 30.If omitted, only a single handle is created.	✓

Slider events

As with the interactions, most of the jQuery UI widgets trigger custom events when interesting things happen to them. We can establish handlers for these events in one of two ways. We can bind handlers in the customary fashion at any point in the ancestor hierarchy, or we can specify the handler as an option, which is what we saw in the previous section.

For example, we might want to handle sliders' slide events in a global fashion on the body element:

```
$('body').bind('slide',function(event,info){……});
```

This allows us to handle slide events for all sliders on the page using a single handler. If the handler is specific to an instance of a slider, we might use the slide option instead when we create the slider:

```
$('#slider').slider({slide:function(event,info){ ... } });
```

This flexibility allows us to establish handlers in the way that best suits our pages.

As with the interaction events, each event handler is passed two parameters: the event instance, and a custom object containing information about the control. Isn't consistency wonderful?

The custom object contains the following properties:

handle—A reference to the <a> element for the handle that's been moved.

value—The current value represented by the handle being moved. For single-handle sliders, this is considered the value of the slider.

values—An array of the current values of all sliders; this is only present for multi-handled sliders.

In the Sliders Lab, the value and values properties are used to keep the value display below the slider up to date. The events that sliders can trigger are summarized in table 11.3. The slidechange event is likely to be the one of most interest because it can be used to keep track of the slider's value or values. Let's say that we have a single-handled slider whose value needs to be submitted to the server upon form submission. Let's also suppose that a hidden input with a name

Table 11.3 Events for the jQuery UI sliders

Event	Option	Description
slide slidechange slidestart slidestop	slide change start stop	Triggered for mousemove events whenever the handle is being dragged through the trough. Returning false cancels the slide. Triggered whenever a handle's value changes, either through user action or programmatically. Triggered when a slide starts. Triggered when a slide stops.

of sliderValue is to be kept up to date with the slide value so that when the enclosing form is submitted, the slider's value acts like just another form control. We could establish an event on the form as follows:

```
$('form').bind('slidechange',function(event,info){  $('[name="sliderValue"]').val(info.value);
});
```

Here's a quick exercise for you to tackle:

Exercise 1—The preceding code is fine as long as there is only one slider in the form. Change the preceding code so that it can work for multiple sliders. How would you identify which hidden input element corresponds to the individual slider controls?

Now let's add some style to our sliders.

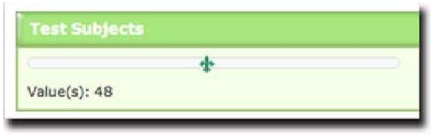
11.2.3 Styling tips for sliders

When an element is transformed into a slider, the class ui-slider is added to it. Within this element, <a> elements will be created to represent the handles, each of which will be given the ui-slider-handle class. We can use these class names to augment the styles of these elements as we choose.

> TIP Can you guess why anchor elements are used to represent the handles? Time's up—it's so that the handles are focusable elements. In the Sliders Lab, create a slider and set focus to a handle by clicking upon it. Now use the left and right arrow keys and see what happens.

Another class that will be added to the slider element is either ui-slider-horizontal or ui-slider-vertical, depending upon the orientation of the slider. This is a useful hook we can use to adjust the style of the slider based upon orientation. In the Sliders Lab, for example, you'll find the following style rules, which adjust the dimensions of the slider as appropriate to its orientation:

```
.testSubject.ui-slider-horizontal {  width: 320px;  height: 8px;
}
```




```
.testSubject.ui-slider-vertical {  height: 108px;  width: 8px;
}
```

The class name testSubject is the class that's used within the Lab to identify the element to be transformed into the slider.

Figure 11.7 With a PNG image and CSS magic, we can make the slider handle a little look like whatever we want.

Here's another neat tip: let's suppose that in order to match the rest of our site, we'd like the slider handler to look like a *fleur-de-lis*. With an appropriate image and a little CSS magic, we can make that happen.

In the Sliders Lab, reset everything, check the checkbox labeled Use Image Handle, and click Apply. The slider looks as shown in figure 11.7.

Here's how it was done. First, a PNG image with a transparent background and containing the *fleur-de-lis* was created, named handle.png. 18 by 18 pixels seems like a good size. Then the following style rule was added to the page:

```
.testSubject a.ui-slider-handle.fancy {  background: transparent url('handle.png') no-repeat 0 0;  border-width: 0;
}
```

Finally, after the slider was created, the fancy class was added to the handle.

```
$('.testSubject .ui-slider-handle').addClass('fancy');
```

One last tip: if you create a range element via the range option, you can style it using the ui-widget-header class. We do so in the Lab page with this line:

```
.ui-slider .ui-widget-header { background-color: orange; }
```

Sliders are a great way to let users enter numeric values in a range without a lot of aggravation on our part or the user's. Let's take a look at another widget that can help us keep our users happy.

Progress bars

Little irks a user more than sitting through a long operation without knowing whether anything is really happening behind the scenes. Although users are somewhat more accustomed to waiting for things in web applications than in desktop applications, giving them feedback that their data is actually being processed makes for much happier, less anxious users.

It's also beneficial to our applications. Nothing good can come of a frustrated user clicking away on our interface and yelling, "Where's my data!" at the screen. The flurry of resulting requests will at best help to bog down our servers, and at worst can cause problems for the backend code.

When a fairly accurate and deterministic means of determining the completion percentage of a lengthy operation is available, a progress bar is a great way to give the user feedback that something is happening.

When not to use progress bars

Even worse than making the user guess when an operation will complete is lying to them about it.

Progress bars should only be used when a reasonable level of accuracy is possible. It's never a good idea to have a progress bar that reaches 10 percent and suddenly jumps to the end (leading users to believe that the operation may have aborted in midstream), or even worse, to be pegged at 100 percent long before the operation actually completes.

If you can't determine an accurate completion percentage, a good alternative to a progress bar is just some indication that something might take a long time; perhaps a text display along the lines of "Please wait while your data is processed—this may take a few minutes ...", or perhaps an animation that gives the illusion of activity while the lengthy operation progresses.

For the latter, a handy website at http://www.ajaxload.info/ generates GIF animations that you can tailor to match your theme.

Visually, a progress bar generally takes the form of a rectangle that gradually "fills" from left to right with a visually distinct inner rectangle to indicate the completion percentage of an operation. Figure 11.8 shows an example progress bar depicting an operation that's a bit less than half complete.

Figure 11.8 A progress bar shows the completion percentage of an operation by "filling" the control from left to right.

jQuery UI provides an easy-to-use progress bar widget that we can use to let users know that our application is hard at work performing the requested operation. Let's see just how easy it is to use.

Creating progress bars

Not surprisingly, progress bars are created using the progressbar() method, which follows the same pattern that's become so familiar:

Command syntax: progressbar

progressbar(options)
progressbar('disable')
progressbar('enable')
progressbar('destroy')
progressbar('option',optionName,value)
progressbar('value',value)

Transforms the wrapped elements (elements recommended) into a progress bar widget.
Parameters
options (Object) An object hash of the options to be applied to the created progress bars, as described in table 11.4.
'disable' (String) Disables a progress bar.

Command syntax: progressbar *(continued)*	
'enable'	(String) Re-enables a disabled progress bar.
'destroy'	(String) Reverts the elements made into progress bar widgets to their original state.
'option'	(String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a progress bar element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.
optionName	(String) The name of the option (see table 11.4) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.
value	(String|Number) The value to be set for the option identified by the optionName parameter (when used with 'option'), or the value between 0 and 100 to be set for the progress bar (if used with 'value').
'value'	(String) If value is provided, sets that value for the progress bar; otherwise the progress bar's current value is returned.

Returns

The wrapped set, except for the case where a value is returned.

Progress bars are conceptually simple widgets, and this simplicity is reflected in the list of options available for the progressbar() method. There are only two, as shown in table 11.4.

Table 11.4 Options for the jQuery UI progress bars

Option	Description
change	(Function) Specifies a function to be established on the progress bar as an event handler for progressbarchange events. See the description of the progress bar events in table 11.5 for details on the information passed to this handler.
value	(Number) Specifies the initial value of the progress bar; 0, if omitted.

Once a progress bar is created, updating its value is as easy as calling the value variant of the method:

```
$('#myProgressbar').progressbar('value',75);
```

Attempting to set a value greater than 100 will result in the value being set to 100. Similarly, attempting to set a value that's a negative number will result in a value of 0. The options are simple enough, as are the events defined for progress bars.

Progress bar events

The single event defined for progress bars is shown in table 11.5.Catching the progressbarchange event could be useful in updating a text value onthe page that shows the exact completion percentage of the control, or for any otherreason that the page might need to know when the value changes.

Table 11.5 Events for the jQuery UI progress bars

Event	Option	Description
progressbarchange	change	Called whenever the value of the progress bar changes. Two parameters are passed: the event instance, and an empty object. The latter is passed in order to be consistent with the other jQuery UI events, but no information is contained within the object.

The progress bar is so simple—only two options and one event—that a Lab page for this control has not been provided. Rather, we thought we'd create a plugin that automatically updates a progress bar as a lengthy operation progresses.

An auto-updating progress bar plugin

When we fire off an Ajax request that's likely to take longer to process than a normal person's patience will accept, and we know that we can deterministically obtain the completion percentage, it's a good idea to comfort the user by displaying a progress bar.

Let's think of the steps we'd run through to accomplish this: 1 Fire off the lengthy Ajax operation. 2 Create a progress bar with a default value of 0. 3 At regular intervals, fire off additional requests that take the pulse of the lengthy operation and return the completion status. It's imperative that this operation be quick and accurate. 4 Use the result to update the progress bar and any text display of the completion percentage.

Sounds pretty easy, but there are a few nuances to take into account, such as making sure that the interval timer is destroyed at the right time.

DEFINING THE AUTO-PROGRESSBAR WIDGET

As this widget is something that could be generally useful across many pages, and because there are non-trivial details to take into account, creating a plugin that's going to handle this for us sounds like a great idea.

We call our plugin the auto-progressbar, and its method, autoProgressbar(), is defined as follows:

Command syntax: autoProgressbar

```
autoProgressbar(options)
autoProgressbar('stop')
autoProgressbar('destroy')
autoProgressbar('value',value)
```

Transforms the wrapped elements (elements recommended) into a progress bar widget.

Parameters

options (Object) An object hash of the options to be applied to the created progress bars, as described in table 11.6.

Command syntax: autoProgressbar *(continued)*	
'stop'	(String) Stops the auto-progressbar widget from checking the completion status.
'destroy'	(String) Stops the auto-progressbar widget and reverts the elements made into the progress bar widget to their original state.
'value'	(String) If value is provided, sets that value for the progress bar; otherwise the progress bar's current value is returned.
value	(String|Number) A value between 0 and 100 to be set for the progress bar used with the 'value' method.

Returns

The wrapped set, except for the case where a value is returned.

The options that we'll define for our plugin are shown in table 11.6.

Table 11.6 Options for the autoProgressbar() plugin method

Option	Description
pulseUrl	(String) Specifies the URL of a server-side resource that will check the pulse of the backend operation that we want to monitor. If this option is omitted, the method performs no operation. The response from this resource must consist of a numeric value in the range of 0 through 100 indicating the completion percentage of the monitored operation.
pulseData	(Object) Any data that should be passed to the resource identified by the pulseUrl option. If omitted, no data is sent.
interval	(Number) The duration, in milliseconds, between pulse checks. The default is 1000(1 second).
change	(Function) A function to be established as the progressbarchange event handler.

Let's get to it.

CREATING THE AUTO-PROGRESSBAR

As usual, we'll start with a skeleton for the method that follows the rules and practices we laid out in chapter 7. (Review chapter 7 if the following doesn't seem familiar.) In a file named jquery.jqia2.autoprogressbar.js we write this outline:

```
(function($){  $.fn.autoProgressbar =
    function(settings,value) {//implementation will go here
    return this; };})(jQuery);
```

The first thing we'll want to do is check to see if the first parameter is a string or not. If it's a string, we'll use the string to determine which method to process. If it's not a string, we'll assume it's an options hash. So we add the following conditional construct:

```
if (typeof settings === "string") {  // process methods here



}
else {
 // process options here
```

```
}
```

Because processing the options is the meat of our plugin, we'll start by tackling the else part. First, we'll merge the user-supplied options with the set of default options, as follows:

```
settings = $.extend({  pulseUrl: null,  pulseData: null,
    interval: 1000,  change: null
},settings||{});if (settings.pulseUrl == null) return this;
```

As in previous plugins that we've developed, we use the $.extend() function to merge the objects. Note also that we continue with the practice of listing *all* options in the default hash, even if they have a null value. This makes for a nice place to see all the options that the plugin supports.

After the merge, if the pulseUrl option hasn't been specified, we return, performing no operation—if we don't know how to contact the server, there's not much we can do.

Now it's time to actually create the progress bar widget:

```
this.progressbar({value:0,change:settings.change});
```

Remember, within a plugin, this is a reference to the wrapped set. We call the jQuery UI progress bar method on this set, specifying an initial value of 0, and passing on any change handler that the user supplied.

Now comes the interesting part. For each element in the wrapped set (chances are there will only be one, but why limit ourselves?) we want to start an interval timer that will check the status of the lengthy operation using the supplied pulseUrl. Here's the code we use for that:

```
this.each(function(){
```

⟵ Iterates over

```
var bar$ = $(this);
```

B wrapped set C Stores interval

```
bar$.data(
```

⟵ handle on widget

```
    'autoProgressbar-interval',    window.setInterval(function(){
```

D Starts interval timer

```
    $.ajax({
```

⟵ Fires off

```
        url: settings.pulseUrl,
```

E Ajax request

```
        data: settings.pulseData,
        global: false,
```

F Receives

```
        dataType: 'json',
```

completion status

```
        success: function(value){

      if (value != null) bar$.autoProgressbar('value',value);  if (value ==
    100) bar$.autoProgressbar('stop');}     });    },settings.interval));});
```

There's a lot going on here, so let's take it one step at a time.
We want each progress bar that will be created to have its own interval timer. Why a user would want to create multiple auto-progressbars may be beyond us, but it's the jQuery way to let them have their rope. We use the each() method B to deal with each wrapped element separately.
For both readability, as well as for use within closures that we'll later create, we capture the wrapped element in the bar$ variable.
We then want to start the interval timer, but we need to keep in mind that later on we're going to want to stop the timer. So we need to store the handle that identifies the timer somewhere that we can easily get at later. jQuery's data() method comes in handy for this C, and we use it to store the handle on the bar element with a name of autoProgressbar-interval.

A call to JavaScript's window.setInterval() function starts the timer D. To this function we pass an inline function that we want to execute on every tick of the timer, and the interval value that we obtain from the interval option.

Within the timer callback, we fire off an Ajax request E to the URL supplied by the pulseUrl option, with any data supplied via pulseData. We also turn off global events (these requests are happening behind the scenes, and we don't want to confuse the page by triggering global Ajax events that it should know nothing about), and specify that we'll be getting JSON data back as the response.

Finally, in the success callback for the request F, we update the progress bar with the completion percentage (which was returned as the response and passed to the callback). If the value has reached 100, indicating that the operation has completed, we stop the timer by calling our own stop method.
After that, implementing the remaining methods will seem easy. In the *if* part of the high-level conditional statement (the one that checked to see if the first parameter was a string or not), we write this:

```
switch (settings) {
case 'stop':
this.each(function(){
window.clearInterval($(this).data('autoProgressbar-interval'))
});
break;
case 'value':
if (value == null) return this.progressbar('value');
this.progressbar('value',value);
break;
case 'destroy':
this.autoProgressbar('stop');
this.progressbar('destroy');
break;
default:
break;
}
```

In this code fragment, we switch to different processing algorithms based on the string in the settings parameter B, which should contain one of: stop, value, or destroy.

For stop we want to kill off all the interval timers that we created for the elements in the wrapped set C. We retrieve the timer handle, which we conveniently stored as data on the element, and pass it to the window.clearInterval() method to stop the timer.
If the method was specified as value, we simply pass the value along to the value method of the progress bar widget.
When destroy is specified, we want to stop the timer, so we just call our own stop method (why copy and paste the same code twice?), and then we destroy the progress bar.
And we're done! Note how whenever we return from any call to our method, we return the wrapped set so that our plugin can participate in jQuery chaining just like any other chainable method.
The full implementation of this plugin can be found in file chapter11/progressbars/jquery.jqia2.autoprogressbar.js.
Let's now turn our attention to testing our plugin.

TESTING THE AUTO-PROGRESSBAR PLUGIN

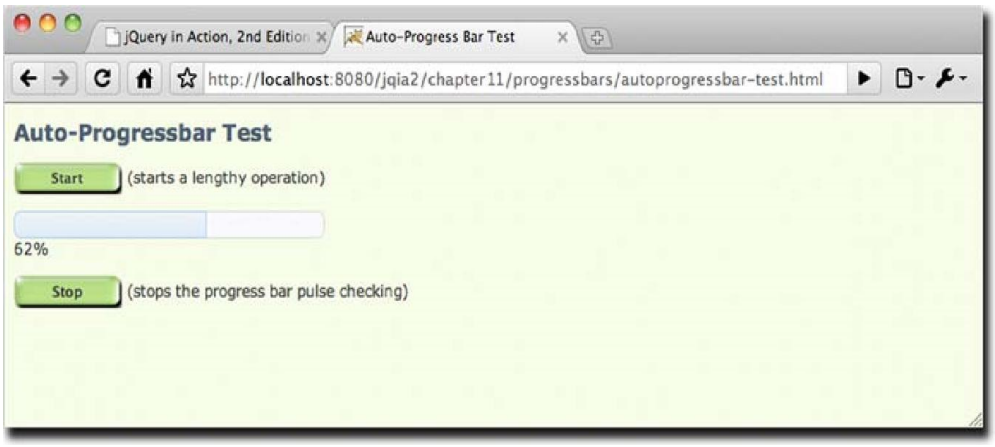
The file chapter11/progressbars/autoprogressbar-test.html contains a test page that uses our new plugin to monitor the completion progress of a long-running Ajax operation. In the interest of saving some space, we won't examine every line of code in that file, but we will concentrated on the portions relevant to using our plugin.
First, let's look at the markup that creates the DOM structures of note:

```
<div>
<button type="button" id="startButton" class="green90x24">Start</button>
(starts a lengthy operation)
</div>
<div>
<div id="progressBar"></div>
<span id="valueDisplay">&mdash;</span>
</div>
<div>
<button type="button" id="stopButton" class="green90x24">Stop</button>
(stops the progress bar pulse checking)
</div>
```

This markup creates four primary elements:

- A Start button that will start a lengthy operation and use our plugin to monitor its progress.
- A to be transformed into the progress bar.
- A to show the completion percentage as text.
- A Stop button that will stop the progress bar from monitoring the lengthy operation.

In action, our test page will look like figure 11.9.



NOTE Because this example uses server-side Ajax operations, it must be run from the Tomcat instance that we set up for the example in chapter 8 (note the 8080 port in the URL). Alternatively, you can run this example remotely by visiting http://www.bibeault.org/jqia2/chapter11/progressbars/autoprogressbar-test.html.

Instrumenting the Start button is the most important operation on this page, and that's accomplished with the following script:

```
$('#startButton').click(function(){
$.post('/jqia2/lengthyOperation',function(){
$('#progressBar')
.autoProgressbar('stop')
.autoProgressbar('value',100);
});
$('#progressBar').autoProgressbar({
pulseUrl: '/jqia2/checkProgress',
change: function(event) {
$('#valueDisplay').text$('#progressBar').autoProgressbar ('value') +
'%');
}
});
});
```

Within the click handler for the Start button, we do two things: kick off the lengthy operation, and create the auto-progressbar.

We post to the URL /jqia2/lengthyOperation, which identifies a process on the server that takes approximately 12 seconds to complete B. We'll get to the success callback in a moment, but first let's skip ahead to the creation of the auto-progressbar.

We call our new plugin D with values that identify a server-side resource, /jqia2/checkProgress, which identifies the process that checks the status of our long-running process and returns the completion percentage as its response. How this is done on the server is completely dependent upon how the backend of the web application is written and that's well beyond the scope of this discussion. (For our example, two separate servlets are used, using the servlet session to keep track of progress.) The

change handler for the progress bar causes the onscreen display of the completion value to be updated.

Now let's backtrack to the success handler for the long-running operation C. When the operation completes, we want to do two things: stop the progress bar, and make sure that the progress bar reflects that the operation is 100 percent done. We easily accomplish this by first calling the stop method of our plugin, followed by a call to the value method. The change handler for the progress bar will update the text

display accordingly.
We've created a really useful plugin using a progress bar. Now let's discuss some styling tips for progress bars.

11.3.4 Styling progress bars

When an element is transformed into a progress bar, the class name ui-progressbar is added to it, and the element created within this element to depict the value is classed with ui-progressbar-value. We can use these class names for CSS rules that augment the style of these elements as we see fit.

For example, you might want to fill the background of the inner element with an interesting pattern, rather than the theme's solid color:

```
.ui-progressbar-value {  background-image: url(interesting-pattern.png);
}
```

Or you could make the progress bar even more dynamic by supplying an animated GIF image as the background image.

Progress bars calm the psyches of our users by letting them know how their opera tions are progressing. Next, let's delight our users by limiting how much they need to type to find what they're looking for.

11.4 Autocompleters

The contemporary acronym *TMI*, standing for "too much information," is usually used in conversation to mean that a speaker has revealed details that are a tad too intimate for the listening audience. In the world of web applications, "too much information" refers not to the nature of the information, but the *amount*.

Although having the vast amount of information that's available on the web at our fingertips is a great thing, it really is possible to have too much information—it's easy to get overwhelmed when fed a deluge of data. Another colloquial expression that describes this phenomenon is "drinking from a fire hose."

When designing user interfaces, particularly those for web applications, which have the ability to access huge amounts of data, it's important to avoid flooding a user with too much data or too many choices. When presenting large data sets, such as report data, good user interfaces give the user tools to gather data in ways that are useful and helpful. For example, filters can be employed to weed out data that isn't relevant to the user, and large sets of data can be paged so that they're presented in digestible chunks. This is exactly the approach taken by our DVD Ambassador example.

As an example, let's consider a data set that we'll be using in this section: a list of DVD titles, which is a data set consisting of 937 titles. It's a large set of data, but still a small slice of larger sets of data (such as the list of all DVDs ever made, for example).

Suppose we wished to present this list to users so that they could pick their favorite flick. We could set up an HTML <select> element that they could use to choose a title, but that would hardly be the friendliest thing to do. Most usability guidelines recommend presenting no more than a dozen or so choices to a user at a time, let alone many hundreds! And usability concerns aside, how practical is it to send such a large data set to the page each time it's accessed by potentially hundreds, thousands, or even millions of users on the web?

jQuery UI helps us solve this problem with an *autocomplete* widget—a control that

acts a lot like a <select> dropdown, but filters the choices to present only those that match what the user is typing into a control.

Creating autocomplete widgets

The jQuery autocomplete widget augments an existing <input> text element to fetch and present a menu of possible choices that match whatever the user types into the input field. What constitutes a match depends on the options we supply to the widget upon creation. Indeed, the autocomplete widget gives us a great deal of flexibility in how to provide the list of possible choices, and how to filter them given the data supplied by the user.

The syntax for the autocomplete() method is as follows:

Command syntax: autocomplete

autocomplete(options)
autocomplete('disable')
autocomplete('enable')
autocomplete('destroy')
autocomplete('option',optionName,value)
autocomplete('search',value)
autocomplete('close')
autocomplete('widget')

Transforms the <input> elements in the wrapped set into an autocomplete control.
Parameters
options (Object) An object hash of the options to be applied to the elements in the wrapped set, as described in table 11.7, making them autocompleters.
'disable' (String) Disables autocomplete controls.
'enable' (String) Re-enables disabled autocomplete controls.
'destroy' (String) Reverts any elements transformed into autocomplete controls to their previous state.
'option' (String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be an autocomplete element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.

Command syntax: autocomplete *(continued)*

optionName	(String) The name of the option (see table 11.7) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.
value	(Object) The value to be set for the option identified by the optionName parameter (when used with 'option'), or the search term (if used with 'search').
'search'	(String) Triggers a search event using the specified value, if present, or the content of the control. Supply an empty string to see a menu of all possibilities.
'close'	(String) Closes the autocomplete menu, if open.
'widget'	(String) Returns the autocomplete element (the one annotated with the ui-autocomplete class name).

Returns

The wrapped set, except for the case where an option, element, search result, or handle value is returned.

For such a seemingly complex control, the list of options available for autocomplete controls is rather sparse, as described in table 11.7.

Option	Description	In Lab?
change	(Function) Specifies a function to be established on the autocompleters as an event handler for autocompletechange events. See the description of the autocomplete events in table 11.8 for details on the information passed to this handler.	✓
close	(Function) Specifies a function to be established on the autocompleters as an event handler for autocompleteclose events. See the description of the autocomplete events in table 11.8 for details on the information passed to this handler.	✓
delay	(Number) The number of milliseconds to wait before trying to obtain the matching values (as specified by the source option). This can help reduce thrashing when non-local data is being obtained by giving the user time to enter more characters before the search is initiated. If omitted, the default is 300 (0.3 seconds).	✓
disabled	(Boolean) If specified and true, the widget is initially disabled.	
focus	(Function) Specifies a function to be established on the autocompleters as an event handler for autocompletefocus events. See the description of the autocomplete events in table 11.8 for details on the information passed to this handler.	✓
minLength	(Number) The number of characters that must be entered before trying to obtain the matching values (as specified by the source option). This can prevent too large a value set from being presented when a few characters isn't enough to whittle the set down to a reasonable level. The default value is 1 character.	✓
open	(Function) Specifies a function to be established on the autocompleters as an event handler for autocompleteopen events. See the description of the autocomplete events in table 11.8 or details on the information passed to this handler.	✓

Option	Description	In Lab?
search select source	(Function) Specifies a function to be established on the autocompleters as an event handler for autocompletesearch events. See the description of the autocomplete events in table 11.8 for details on the information passed to this handler. (Function) Specifies a function to be established on the autocompleters as an event handler for autocompleteselect events. See the description of the autocomplete events in table 11.8 for details on the information passed to this handler. (String|Array|Function) Specifies the manner in which the data that matches the input data is obtained. A value must be provided or the autocomplete widget won't be created. This value can be a string representing the URL of a server resource that will return matching data, an array of local data from which the value will be matched, or a function that serves as a general callback from providing the matching values. See section 11.4.2 for more information on this option.	✓ ✓ ✓

As you might have guessed, an Autocompleters Lab (shown in figure 11.10) has been provided. Load it from chapter11/autocompleters/lab.autocompleters.html and follow along as you review the options.

NOTE In this Lab, the URL variant of the source option requires the use of server-side Ajax operations. It must be run from the Tomcat instance we set up for the example in chapter 8

(note the 8080 port in the URL). Alternatively, you can run this example remotely by visiting http://www.bibeault.org/ jqia2/chapter11/autocompleters/lab.autocompleters.html.

Except for source, these options are all fairly self-explanatory. Leaving the source option at its default setting, use the Autocompleters Lab to observe the events that transpire and the behavior of the minLength and delay options until you feel that you have grasped them.

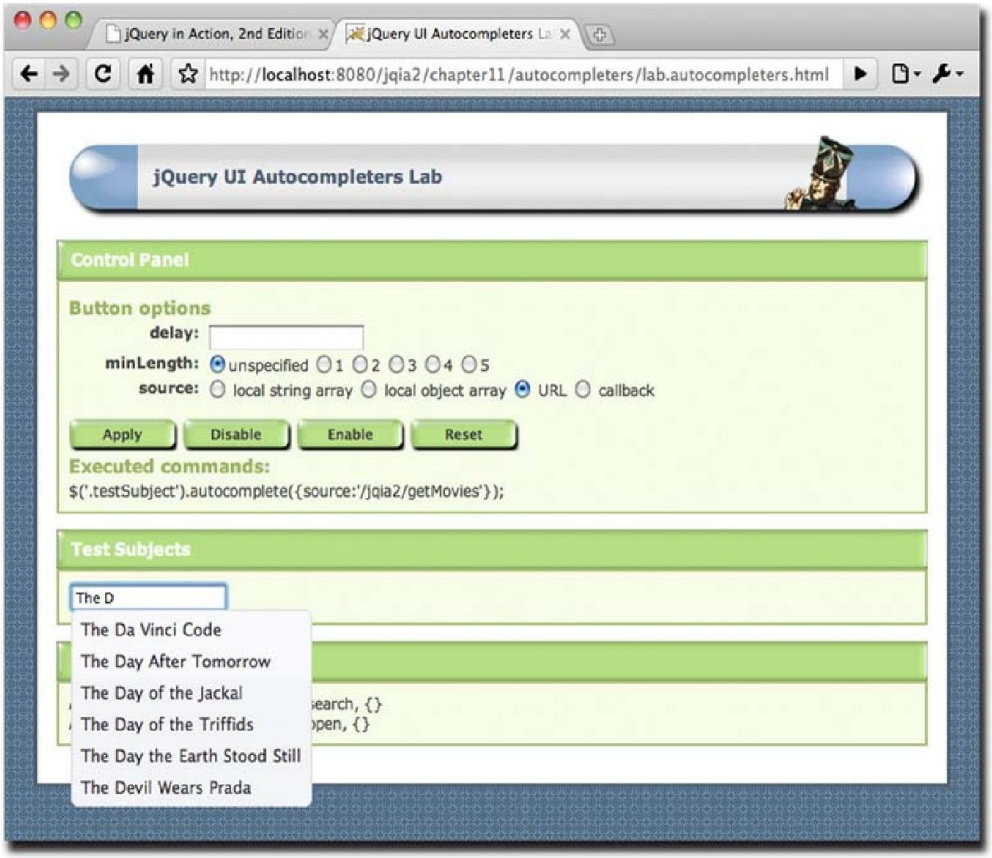
Now let's see what it takes to provide source data for this widget.

Autocomplete sources

The autocomplete widget gives us a lot of flexibility for providing the data values that match whatever the user types in. Source data for the autocompleters takes the form of an array of candidate items, each of which has two properties:

. A value property that represents the actual values. These are the strings that are matched against as the user types into the control, and they're the values that will be injected into the control when a menu item is selected.

. A label property that represents the value, usually as a shorter form. These strings are what is displayed in the autocomplete menu, and they don't participate in the default matching algorithms.



This data can come from a variety of sources.
For cases where the data set is fairly small (dozens, not hundreds or more), the data can be provided as a local array. The following example is taken from the Autocompleters Lab and provides candidate data that uses usernames as labels and full names as the values:

```
var sourceObjects = [
{ label: 'bear', value: 'Bear Bibeault'},
{ label: 'yehuda', value: 'Yehuda Katz'},
{ label: 'genius', value: 'Albert Einstein'},
{ label: 'honcho', value: 'Pointy-haired Boss'},
{ label: 'comedian', value: 'Charlie Chaplin'}
];
```

When displayed, the labels (usernames) are what appear in the autocomplete menu, but matching is performed on the *values* (full names), and the value is what is set into the control upon selection.
This is handy when we want to represent longer data with shorter values in the menus, but for many cases, perhaps even most, the label and the value will be the same. For these common cases, jQuery UI lets us specify the data as an array of strings, and takes the string value to be both the label and the value.

TIP When providing objects, if only one of label or value is specified, the provided data is automatically used for both label and value.

The entries don't have to be in any particular order (such as sorted) for the widget to work correctly, and matching entries will be displayed in the menu in the order that they appear within the array.
When local data is used, the matching algorithm is such that any candidate value that contains what the user has typed, called the *term*, is deemed to match. If this isn't what you want—let's say you only want to match values that *begin* with the term—fear not! There are two more-general ways to supply the source data that give us complete control over the matching algorithm.
For the first of these schemes, the source can be specified as the URL of a serverside resource that returns a response containing the data values that match the term, which is passed to the resource as a request parameter named term. The returned data should be a JSON response that evaluates to one of the formats supported for local data, usually an array of strings.
Note that this variant of source is expected to perform the search and return *only* the matching elements—no further processing of the data will take place. Whatever values are returned are displayed in the autocomplete menu.
When we need maximum flexibility, another scheme can be used: a callback function can be supplied as the source option, and it's called whenever data is needed by the widget. This callback is invoked with two parameters:

- An object with a single property, term, that contains the term to be matched.
- A callback function to be called, which is passed the matching results to be displayed.

This set of results can be either of the formats accepted as local data, usually an array of strings.
This callback mechanism offers the most flexibility, because we can use whatever mechanisms and algorithms we want to turn the term into a set of matching elements.
A skeleton for how to use this variant of source is as follows:

```
$('#control').autocomplete({
source: function(request,response) {
var term = request.term;
var results;
```

```
// algorithm here to fill results array
response(results);
}
});
```

Play around with the source options in the Autocompleters Lab. A few things to note about the different source options in the Lab:
- The local string option provides a list of 79 values, all of which begin with the letter F.
- The local object option provides a short list of usernames for labels, and full names as values. Note how the matching occurs on the *values*, not the labels. (Hint: enter the letter *b*.)
- For the URL variant, the backend resource only matches values that *begin* with the term. It uses a different algorithm than when local values are supplied (in which the term can appear anywhere within the string). This difference is intentional and is intended to emphasize that the backend resource is free to employ whatever matching criteria it likes.
- The callback variant simply returns the entire value set of 79 F-titles provided by the local option. Make a copy of the Lab page, and modify the callback to play around with whatever algorithm you'd like to filter the returned values.

Various events are triggered while an autocomplete widget is doing its thing. Let's see what those are.

Play around with the source options in the Autocompleters Lab. A few things to note about the different source options in the Lab:

. The local string option provides a list of 79 values, all of which begin with the letter F.

. The local object option provides a short list of usernames for labels, and full names as values. Note how the matching occurs on the *values*, not the labels. (Hint: enter the letter *b*.)

. For the URL variant, the backend resource only matches values that *begin* with the term. It uses a different algorithm than when local values are supplied (in which the term can appear anywhere within the string). This difference is intentional and is intended to emphasize that the backend resource is free to employ whatever matching criteria it likes.

. The callback variant simply returns the entire value set of 79 F-titles provided by the local option. Make a copy of the Lab page, and modify the callback to play around with whatever algorithm you'd like to filter the returned values.

Various events are triggered while an autocomplete widget is doing its thing. Let's see what those are.

Autocomplete events

During an autocomplete operation, a number of custom events are triggered, not only to inform us of what's going on, but to give us a chance to cancel certain aspects of the operation.

As with other jQuery UI custom events, two parameters are passed to the event handlers: the event and a custom object. This custom object is empty except for autocompletefocus, autocompletechange, and autocompleteselect events. For the focus, change, and select events, this object contains a single property named item, which in turn contains the properties label and value, representing the label and value of the focused or selected value. For all the event handlers, the function context (this) is set to the <input> element.

Table 11.8 Events for the jQuery UI autocompleters

Event	Option	Description
autocompletechange	change	Triggered when the value of the <input> element is changed based upon a selection. When triggered, this event will always come after the autocompleteclose event is triggered.
autocompleteclose	close	Triggered whenever the autocomplete menu closes.
autocompletefocus	focus	Triggered whenever one of the menu choices receives focus. Unless canceled (for example, by returning false), the focused value is set into the <input> element.

 Table 11.8 Events for the jQuery UI autocompleters *(continued)*

Event	Option	Description
autocompleteopen autocompletesearch autocompleteselect	open search select	Triggered after the data has been readied and the menu is about to open. Triggered after any delay and minLength criteria have been met, just before the mechanism specified by source is activated. If canceled, the search operation is aborted. Triggered when a value is selected from the autocomplete menu. Canceling this event prevents the value from being set into the <input> element (but doesn't prevent the menu from closing).

The Autocompleters Lab uses all of these events to update the console display as the events are triggered. Now let's take a look at dressing up our autocompleters.

Autocompleting in style

As with the other widgets, autocompleters inherit style elements from the jQuery UI CSS theme via the assignment of class names to the elements that compose the autocompleter. When an <input> element is transformed into an autocompleter, the class uiautocomplete-input is added to it.

When the autocomplete menu is created, it's created as an unordered list element (<ul>) with class names ui-autocomplete and ui-menu. The values within the menu are created as <li> elements with class name ui-menu-item. And within those list items, anchor elements are created that get the ui-state-hover class when hovered over.

We can use these classes to hook our own styles onto the autocomplete elements. For example, let's say that we want to give the autocomplete menu a slight level of transparency. We could do that with this style rule:

```
.ui-autocomplete.ui-menu { opacity: 0.9; }
```

Be careful with that. Make it *too* transparent and it becomes unreadable.
The autocomplete menu can end up pretty big if there are lots of matches. If we'd like to fit more entries in less space, we can shrink the font size of the entries with a rule like this:

```
.ui-autocomplete.ui-menu .ui-menu-item { font-size: 0.75em; }
```

Note that ui-menu-item isn't a class name specific to the autocompleter (if it were, it would have the text autocomplete within it), so we qualify it with ui-autocomplete and ui-menu to make sure we don't inadvertently apply the style to other elements on the page

What if we really wanted to make hovered items stand out? We could change their border to red:

```
.ui-autocomplete.ui-menu a.ui-state-hover { border-color: red; }
```

Autocompleters let us let our users hone down large datasets quickly, preventing information overload. Now let's see how we can simplify yet another long-standing pain point in data entry: dates.

Date pickers

Entering date information has been another traditional source of anxiety for web developers and frustration for end users. A number of approaches have been tried using the basic HTML 4 controls, all of which have their drawbacks.

Many sites will present the user with a simple text input into which the date must be entered. But even if we include instructions such as, "Please enter the date in dd/ mm/yyyy format", people still tend to get it wrong. And so, apparently, do some web developers. How many times have you wanted to throw your computer across the room upon discovering, after 15 failed attempts, that you had to include leading zeroes when entering a single digit date or month value?

Another approach uses three dropdowns, one each for month, day, and year. Although this vastly reduces the possibility of user error, it's clumsy and requires a lot of clicks to choose a date. And developers still need to guard against entries such as February 31.

When people think of dates, they think of calendars, so the most natural way to have them enter a date is to let them pick it from a calendar display.

Frequently called *calendar controls* or *date pickers*, scripts to create these controls have been around for some time, but they've generally been cantankerous to configure, and awkward to use on pages, including trying to match styling. Leave it to jQuery and jQuery UI to make it easy with jQuery UI datepickers.

Creating jQuery datepickers

Creating a jQuery datepicker is easy, especially if you take the default values. It may only seem complex because there are lots of options for configuring the datepicker in the manner that best suits our applications.

As with other jQuery UI elements, the datepicker() exposes the basic set of UI methods and also offers some specific methods to control the element after creation:

Command syntax: datepicker

datepicker(options)
datepicker('disable')
datepicker('enable')
datepicker('destroy')
datepicker('option',optionName,value)
datepicker('dialog',dialogDate,onselect,options,position)
datepicker('isDisabled')
datepicker('hide',speed)
datepicker('show')
datepicker('getDate')
datepicker('setDate',date)
datepicker('widget')

Transforms the <input>, , and elements in the wrapped set into a datepicker control. For <input> elements, the datepickeris displayed on focus; for other elements, creates an inline datepicker.

Parameters
options (Object) An object hash of the options to be applied to the elements in the wrapped set, as described in table 11.9, making them datepickers.
'disable' (String) Disables datepicker controls.
'enable' (String) Re-enables disabled datepicker controls.
'destroy' (String) Reverts any elements transformed into datepicker controls to their previous state.
'option' (String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a datepicker element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.
optionName (String) The name of the option (see table 11.9) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.
value (Object) The value to be set for the option identified by the optionName parameter.
'dialog' (String) Displays a jQuery UI dialog box containing a datepicker.
dialogDate (String|Date) Specifies the initial date for the datepicker in the dialog box as a string in the current date format (see the description of the dateFormat option in table 11.9) or a Date instance.
onselect (Function) If specified, defines a callback to be invoked with the date text and datepicker instance when a date is selected.
position (Array|Event) An array specifying the position of the dialog box as [left,top], or a mouseevent Event instance from which the position will be determined.
If omitted, the dialog box is centered in the window.
'isDisabled' (String) Returns true or false reporting whether the datepicker is currently disabled or not.
'hide' (String) Closes the datepicker.

Command syntax: datepicker *(continued)*

speed (String|Number) One of slow, normal, or fast, or a value in milliseconds that controls the animation closing the datepicker. 'show' (String) Opens the datepicker. 'getDate' (String) Returns the currently selected date for the datepicker. This value can be null if no value has yet been selected. 'setDate' (String|Date) Sets the specified date as the current date of the datepicker.

date(String|Date) Sets the date for the datepicker. This value can be a Date instance, or a string that identifies an absolute or relative date. Absolute dates are specified using the date format for the control (specified by the dateFormat option, see table 11.9), or a string of values specifying a date relative to today. The values are numbers followed by m for month, d for day, w for week, and y for year. For example, tomorrow is +1d, and a week and a half could be +1w+4d. Both positive and negative values can be used.

widget'(String) The datapicker widget element; the one annotated with the ui-datapicker class name.

Returns

The wrapped set, except for the cases where values are returned, as described above.

Seemingly to make up for the Spartan set of options available for autocompleters, datepickers offer a dizzying array of options that make it the most configurable widget in the jQuery UI set. Don't get too overwhelmed; frequently the defaults are just what we want. But the options are there in case we need to change the way the datepicker works to better fit into our sites.

But all those options do make for a rather complicated Datepickers Lab page—as shown in figure 11.11. You'll find it in file chapter11/datepickers/lab.datepickers.html.

As you work your way through the generous set of options described in table 11.9, try them out in the Datepickers Lab.

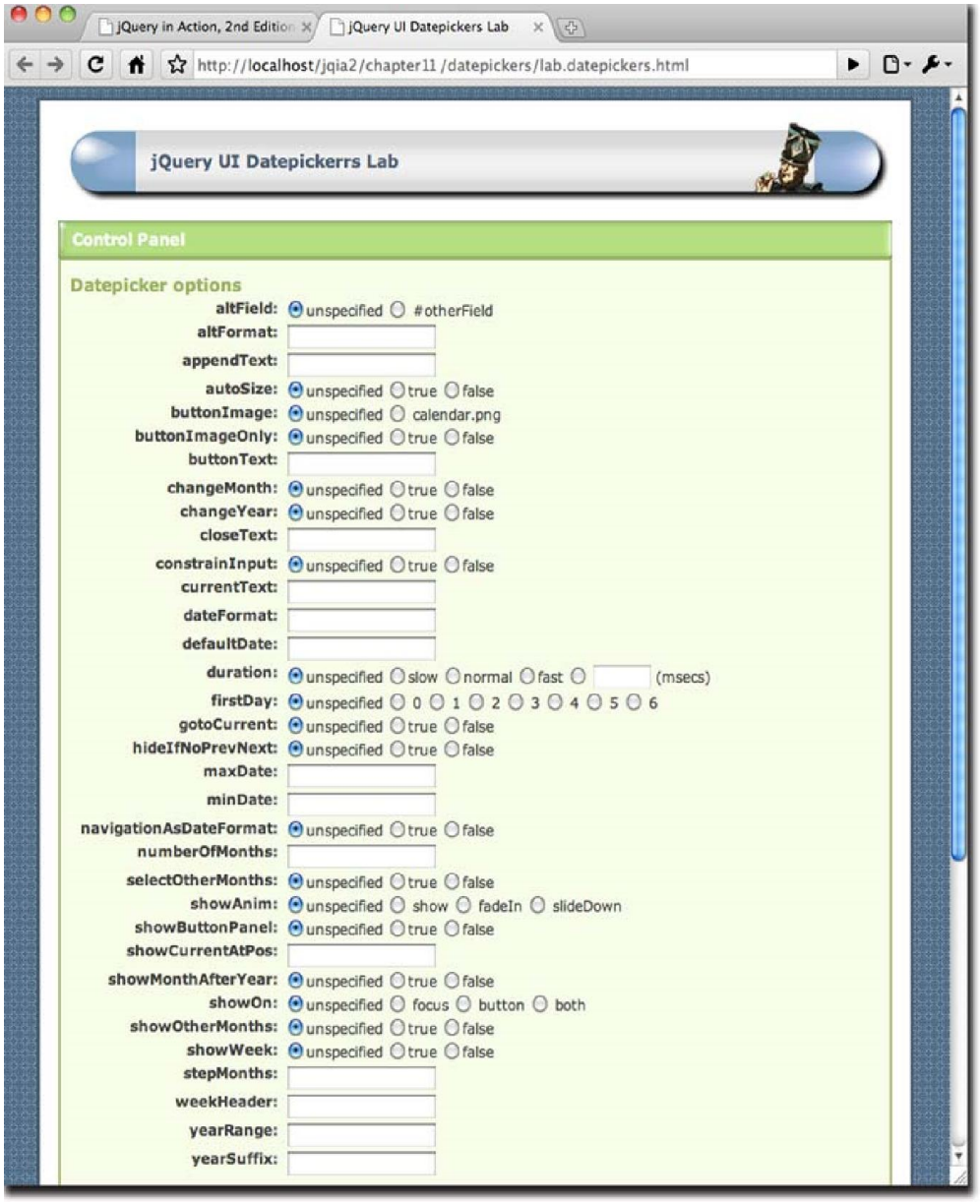

jQuery in Action, 2nd Edition
jQuery UI Datepickers Lab
http://localhost/jqia2/chapter11 /datepickers/lab.datepickers.html
jQuery UI Datepickerrs Lab
Control Panel
Datepicker options
altField: unspecified #otherField
altFormat:
appendText:
autoSize: unspecified true false
buttonImage: unspecified calendar.png
buttonImageOnly: unspecified true false
buttonText:
changeMonth: unspecified true false
changeYear: unspecified true false
closeText:
constrainInput: unspecified true false
currentText:
dateFormat:
defaultDate:
duration: unspecified slow normal fast (msecs)
firstDay: unspecified 0 1 2 3 4 5 6
gotoCurrent: unspecified true false
hideIfNoPrevNext: unspecified true false
maxDate:
minDate:
navigationAsDateFormat: unspecified true false
numberOfMonths:
selectOtherMonths: unspecified true false
showAnim: unspecified show fadeIn slideDown
showButtonPanel: unspecified true false
showCurrentAtPos:
showMonthAfterYear: unspecified true false
showOn: unspecified focus button both
showOtherMonths: unspecified true false
showWeek: unspecified true false
stepMonths:
weekHeader:
yearRange:
yearSuffix:

Table 11.9 Options for the jQuery UI datepickers

Option	Description	In Lab?
`altField`	(Selector) Specifies a jQuery selector for a field that's to also be updated with any date selections. The `altFormat` option can be used to set the format for this value. This is quite useful for setting date values into a hidden input element to be submitted to the server, while displaying a more user-friendly format to the user.	✓
`altFormat`	(String) When an `altField` is specified, provides the format for the value to be written to the alternate element. The format of this value is the same as for the `$.datepicker.formatDate()` utility function—see its description in section 11.5.2 for details.	✓
`appendText`	(String) A value to be placed after the `<input>` element, intended to show instructions to the user. This value is displayed within a `<span>` element created with the class name `ui-datepicker-append`, and can contain HTML markup.	✓
`autoSize`	(Boolean) If `true`, the size of the `<input>` element is adjusted to accommodate the datepicker's date format as set with the `dateFormat` option. If omitted, no resize takes place.	✓
`beforeShow`	(Function) A callback that's invoked just before a datepicker is displayed, with the `<input>` element and datepicker instance passed as parameters. This function can return an options hash used to modify the datepicker.	✓
`beforeShowDay`	•(Function) A callback that's invoked for each day in the datepicker just before it's displayed, with the date passed as the only parameter. This can be used to override some of the default behavior of the day elements. This function must return a three-element array, as follows: • [0]—`true` to make the date selectable, `false` otherwise • [1]—A space-delimited string of CSS class names to be applied, or an empty string to apply none • [2]—An optional string to apply a tooltip to the day element	
`buttonImage`	(String) Specifies the path to an image to be displayed on the button enabled by setting the `showOn` option to one of `button` or `both`. If `buttonText` is also provided, the button text becomes the `alt` attribute of the button.	✓
`buttonImageOnly`	(Boolean) If `true`, specifies that the image specified by `buttonImage` is to appear standalone (not on a button). The `showOn` option must still be set to one of `button` or `both` for the image to appear.	✓
`buttonText`	(String) Specifies the caption for the button enabled by setting the `showOn` option to one of `button` or `both`. If `buttonImage` is also specified, this text becomes the `alt` attribute of the image.	✓
`calculateWeek`	(Function) A custom function to calculate and return the week number for a date passed as the lone parameter. The default implementation is that provided by the `$.datepicker.iso8601Week()` utility function.	

Option	Description	In Lab?
changeMonth	(Boolean) If true, a month dropdown is displayed, allowing the user to directly change the month without using the arrow buttons to step through them. If omitted, no dropdown is displayed.	✓
changeYear	(Boolean) If true, a year dropdown is displayed, allowing the user to directly change the year without using the arrow buttons to step through them. If omitted, no dropdown is displayed.	✓
closeText	(String) If the button panel is displayed via the showButtonPanel option, specifies the text to replace the default caption of Done for the close button.	✓
constrainInput	(Boolean) If true (the default), text entry into the <input> element is constrained to characters allowed for the date format of the control (see dateFormat).	✓
currentText	(String) If the button panel is displayed via the showButtonPanel option, specifies the text to replace the default caption of Today for the current button.	✓
dateFormat	(String) Specifies the date format to be used. See section 11.5.2 for details.	✓
dayNames	(Array) A 7-element array providing the full day names with the 0th element representing Sunday. Can be used to localize the control. The default set is the full day names in English.	
dayNamesMin	(Array) A 7-element array providing the minimal day names with the 0th element representing Sunday, used as column headers. Can be used to localize the control. The default set is the first two letters of the English day names.	
dayNamesShort	(Array) A 7-element array providing the short day names with the 0th element representing Sunday. Can be used to localize the control. The default set is the first three letters of the English day names.	
defaultDate	(Date|Number|String) Sets the initial date for the control, overriding the default value of today, if the <input> element has no value. This can be a Date instance, the number of days from today, or a string specifying an absolute or relative date. See the description of the date parameter in the method syntax for the datapicker() method for more details.	✓
disabled	(Boolean) If specified and true, the widget is initially disabled.	
duration	(String|Number) Specifies the speed of the animation that makes the datepicker appear. Can be one of slow, normal, (the default) or fast, or the number of milliseconds for the animation to span.	✓
firstDay	(Number) Specifies which day is considered the first day of the week, and will be displayed as the left-most column. Sunday (the default) is 0, Saturday is 6.	✓
gotoCurrent	(Boolean) If true, the current day link is set to the selected date, overriding the default of today.	✓

Table 11.9 Options for the jQuery UI datepickers *(continued)*

Option	Description	In Lab?
`hideIfNoPrevNext`	(Boolean) If `true`, hides the next and previous links (as opposed to merely disabling them) when they aren't applicable, as determined by the settings of the `minDate` and `maxDate` options. Defaults to `false`.	✓
`isRTL`	(Boolean) If `true`, the localizations specify a right-to-left language. Used by localized version of this control. Defaults to `false`.	
`maxDate`	(Date|Number|String) Sets the maximum selectable date for the control. This can be a `Date` instance, the number of days from today, or a string specifying an absolute or relative date. See the description of the `date` parameter in the datepicker `setDate` method syntax for more details.	✓
`minDate`	(Date|Number|String) Sets the minimum selectable date for the control. This can be a `Date` instance, the number of days from today, or a string specifying an absolute or relative date. See the description of the `date` parameter in the method syntax for the `datepicker()` method for more details.	✓
`monthNames`	(Array) A 12-element array providing the full month names with the 0th element representing January. Can be used to localize the control. The default set is the full month names in English.	
`monthNamesShort`	(Array) A 12-element array providing the short month names with the 0th element representing January. Can be used to localize the control. The default set is the first three letters of the English month names.	✓
`navigationAsDateFormat`	(Boolean) If `true`, the navigation links for `nextText`, `prevText`, and `currentText` are passed through the `$.datepicker.formatDate()` function prior to display. This allows date formats to be supplied for those options that get replaced with the relevant values. Defaults to `false`.	✓
`nextText`	(String) Specifies the text to replace the default caption of Next for the next month navigation link. Note that the ThemeRoller replaces this text with an icon.	✓
`numberOfMonths`	(Number|Array) The number of months to show in the datepicker, or a 2-element array specifying the number of rows and columns for a grid of months. For example, `[3,2]` will display 6 months in a 3-row by 2-column grid. By default, a single month is shown.	✓
`onChangeMonthYear`	(Function) A callback that's invoked when the datepicker moves to a new month or year, with the selected year, month (1-based), and datepicker instance passed as parameters, and the function context is set to the input field element.	
`onClose`	(Function) A callback invoked whenever a datepicker is closed, passed the selected date as text (the empty string if there is no selection), and the datepicker instance, and the function context is set to the input field element.	

onSelect	(Function) A callback invoked whenever a date is selected, passed the selected date as text (the empty string if there is no selection), and the datepicker instance, and the function context is set to the input field element.	
prevText	(String) Specifies the text to replace the default caption of Prev for the previous month navigation link. (Note that the ThemeRoller replaces this text with an icon.)	✓
selectOtherMonths	(Boolean) If true, days shown before or after the displayed month(s) are selectable. Such days aren't displayed unless the showOtherMonths option is true. By default, the days aren't selectable.	✓
shortYearCutoff	(Number|String) If a number, specifies a value between 0 and 99 years before which any 2-digit year values will be considered to belong to the previous century. For example, if specified as 50, the year 39 would be considered to be 2039, and the year 52 would be inter-preted as 1952. If a string, the value undergoes a numeric conversion and is added to the current year. The default is +10 which represents 10 years from the current year.	
showAnim	(String) Sets the name of the animation to be used to show and hide the datepicker. If specified, must be one of show (the default), fadeIn, slideDown, or any of the jQuery UI show/hide animations.	✓
showButtonPanel	(Boolean) If true, a button panel at the bottom of the datepicker is displayed, containing current and close buttons. The caption of these buttons can be provided via the currentText and closeText options. Defaults to false.	✓
showCurrentAtPos	(Number) Specifies the 0-based index, starting at the upper left, of where the month containing the current date should be placed within a multi-month display. Defaults to 0.	?✓
showMonthAfterYear	(Boolean) If true, the positions of the month and year are reversed in the header of the datepicker. Defaults to false.	✓
showOn	(String) Specifies what triggers the display of the datepicker as one of focus, button, or both. focus (default) causes the datepicker to display when the <input> element gains focus, whereas button causes a button to be created after the <input> element (but before any appended text) that trig-gers the datepicker when clicked. The button's appearance can be var-ied with the buttonText, buttonImage, and buttonImageOnly options. bothcauses the trigger button to be created, and for focus events to also trigger the datepicker.	✓
showOptions	(Object) When a jQuery UI animation is specified for the showAnim option, provides an option hash to be passed to that animation.	

showOtherMonths	(Boolean) If true, dates before or after the first and last days of the current month are displayed. These dates aren't selectable unless the selectOtherMonths option is also set to true. Defaults to false.	✓
showWeek	(Boolean) If true, the week number is displayed in a column to the left of the month display. The calculateWeekoption can be used to alter the manner in which this value is determined. Defaults to false.	✓
stepMonths	(Number) Specifies how many months to move when one of the month navigation controls is clicked. By default, a single month is stepped.	✓
weekHeader	(String) The text to display for the week number column, overriding the default value of Wk, when showWeek is true.	✓
yearRange	(String) When changeYear is true, specifies limits on which years are displayed in the dropdown in the form from:to. The values can be absolute or relative (for example: 2005:+2, for 2005 through 2 years from now). The prefix c can be used to make relative values off-set from the selected year rather than the current year (example: c-2:c+3).	✓
yearSuffix	(String) Text that's displayed after the year in the datepicker header.	✓

Datepicker date formats

A number of the datepicker options listed in table 11.9 employ a string that represents a *date format*. These are strings that specify a pattern for formatting and parsing dates. Character patterns within the string represent parts of dates (for example, y for year, and MM for full month name) or simply template (literal) text.
Table 11.10 shows the character patterns used within date format patterns and what they represent.

d Date within month without leading zeroes
dd 2-digit date within month with leading zeroes for values less than 10

- Day of the year without leading zeroes

oo 3-digit day within the year with leading zeroes for values less than 100
D Short day name
DD Full day name
m Month of the year with no leading zeroes, where January is 1
mm 2-digit month within the year with leading zeroes for values less than 10
M Short month name
MM Full month name
y 2-digit year with leading zeroes for values less than 10
yy 4-digit year
@ Number of milliseconds since January 1, 1970
! Number of 100 ns ticks since January 1, year 1
'' Single quote character
'...' Literal text (quoted with single quotes)
Anything else Literal text

$.datepicker.ATOM yy-mm-dd
$.datepicker.COOKIE D, dd M yy
$.datepicker.ISO_8601 yy-mm-dd
$.datepicker.RFC_822 D, d M y
$.datepicker.RFC_850 DD, dd-M-y
$.datepicker.RFC_1036 D, d M y
$.datepicker.RFC_1123 D, d M yy
$.datepicker.RFC_2822 D, d M yy
$.datepicker.RSS D, d M y
$.datepicker.TICKS !
$.datepicker.TIMESTAMP @
$.datepicker.W3C yy-mm-dd Table 11.11 Date format
pattern constants

Datepicker events

Surprise! There aren't any!

The datepicker code in jQuery UI 1.8 is some of the oldest in the code base, and it hasn't been updated to adhere to the modern event-triggering conventions that the other widgets follow. Expect this to change in a future version of jQuery UI, to the point that the jQuery UI roadmap (which you can find at http://wiki.jqueryui.com/ Roadmap) states that the widget will be completely rewritten for version 2.0.

For now, the options that allow us to specify callbacks when interesting things happen to a datepicker are beforeShow, beforeShowDay, onChangeMonthYear, onClose, and onSelect. All the callbacks invoked via these options have the <input> elements set as their function contexts.

Although datepickers may lack the event triggering that other widgets sport, they do give us some extras: a handful of useful utility functions. Let's see what those can do for us.

11.5.4 Datepicker utility functions

Dates can be cantankerous data types. Just think of the nuances of dealing with years and leap years, months of differing lengths, weeks that don't divide into months evenly, and all the other oddities that plague date information. Luckily for us, the JavaScript Date implementation handles most of those details for us. But there are a few areas where it falls short—the formatting and parsing of date values being two of them.

The jQuery UI datepicker steps up to the plate and fills in those gaps. In the guise of utility functions, jQuery UI provides the means to not only format and parse date values, but also to make the large number of datepicker options a bit easier to handle for pages with more than one datepicker.

Let's start there.

SETTING DATEPICKER DEFAULTS

When our datepickers need to use multiple options to get the look and behavior we want, it seems just plain wrong to cut and paste the same set of options for every datepicker on the page. We could store the options object in a global variable and reference it from every datepicker creation, but jQuery UI lets us go one better by providing a means to simply register a set of default options that supersedes the defined defaults. This utility function's syntax is as follows:

Command syntax: $.datepicker.setDefaults

$.datepicker.setDefaults(options)

Sets the options passed as the defaults for all subsequently created datepickers.

Parameters

options	(Object) An object hash of the options to be used as the defaults for all datepickers.

Returns

Nothing.

As you'll recall from the list of datepicker options, some of the options specify formats for how date values are to be displayed. That's a useful thing to be able to do in general, and jQuery UI makes it available directly to us.

FORMATTING DATE VALUES

We can format any date value using the $.datepicker.formatDate()utility function, defined as follows:

Command syntax: $.datepicker.formatDate

$.datepicker.formatDate(format,date,options)

Formats the passed date value as specified by the passed format pattern and options.

Parameters

format	(String) The date format pattern string as described in tables 11.10 and 11.11.
date	(Date) The date value to be formatted.
options	(Object) An object hash of options that supply alternative localization values for day and month names. The possible options are dayNames, dayNamesShort, monthNames, and monthNamesShort. See table 11.9 for details of these options. If omitted, the default English names are used.

Returns

The formatted date string.

That sort of obsoletes the date formatter we set up in chapter 7! But that's OK, we learned a lot from that exercise, and we can always use it in projects that don't use jQuery UI.

What other tricks does the datepicker have up its sleeve for us?

PARSING DATE STRINGS

As useful as formatting date values into text strings is, it's just as useful—if not even more so—to convert text strings into date values. jQuery UI gives us that ability with the $.datepicker.parseDate() function, whose syntax is as follows:

Command syntax: $.datepicker.parseDate

$.datepicker.parseDate(format,value,options)

Converts the passed text value into a date value using the passed format pattern and options.

Parameters

format (String) The date format pattern string as described in tables 11.10 and 11.11.value (String) The text value to be parsed.options (Object) An object hash of options that supply alternative localization values for day and month names, as well as specifying how to handle 2-digit year values. Thepossible options are shortYearCutoff, dayNames, dayNamesShort, monthNames, and monthNamesShort.See table 11.9 for details of these options. If omitted, the default English namesare used, and the rollover year is +10.

Returns

The parsed date value.

There's one more utility function that the datepicker makes available.

GETTING THE WEEK IN THE YEAR

As a default algorithm for the calculateWeek option, jQuery UI uses an algorithm defined by the ISO 8601 standard. In the event that we might have some use for this algorithm outside of a datepicker control, it's exposed to use as the $.datepicker.iso8601Week() function:

Command syntax: $.datepicker.iso8601Week

$.datepicker.iso8601Week(date)

Given a date value, calculates the week number as defined by ISO 8601.

Parameters

date (Date) The date whose week number is to be calculated.

Returns

The computed week number.

The ISO 8601 definition of week numbering is such that weeks start on Mondays, and the first week of the year is the one that contains January 4th (or in other words, the week containing the first Thursday).

We've seen jQuery UI widgets that allow us to gather data from the user in an intuitive manner, so we're now going to turn our attention to widgets that help us organize our content. If your eyes are getting bleary at this point, now might be a good time to sit back for a moment and enjoy a snack; preferably one containing caffeine.

When you're ready, let's forge on ahead to examine one of the most common organization metaphors on the web—tabs.

11.6 Tabs

Tabs probably need no introduction. As a navigation method, they've become ubiquitous on the web,

surpassed only by links themselves. Mimicking physical card index tabs, GUI tabs allow us to quickly flip between sets of content logically grouped at the same level.

In the bad old days, switching between tabbed panels required full-page refreshes, but today we can just use CSS to show and hide elements as appropriate, and even employ Ajax to fetch hidden content on an as-needed basis.

As it turns out "just using CSS" turns out to be a fair amount of work to get right, so jQuery UI gives us a ready-made tabs implementation that, of course, matches the downloaded UI theme.

Creating tabbed content

Most of the widgets we've examined so far take a simple element, such as a <button>, , or <input>, and transforms it into the target widget. Tabs, by nature, start with a more complex HTML construct.

A canonical construct for a tabset with three tabs should follow this pattern:

```
<div id="tabset">
<ul>
<li><a href="#panel1">Tab One</a></li>
<li><a href="#panel2">Tab Two</a></li>
<li><a href="#panel3">Tab Three</a></li>
</ul>
<div id="panel1">
... content ...
</div>
<div id="panel2">
... content ...
</div>
<div id="panel3">
... content ...
</div>
</div>
```

This construct consists of a element that contains the entire tabset B, which consists of two subsections: an unordered list (<ul>) containing list items (<li>) that will become the tabs C, and a set of elements, one for each corresponding panel D.

Each list item that represents a tab contains an anchor element (<a>) that not only defines the association between the tab and its corresponding panel, but also serves as a focusable element. The href attribute of these anchors specifies an HTML anchor hash, useable as a jQuery id selector, for the panel that it's to be associated with. Each tab's content can alternatively be fetched from the server via an Ajax request upon first selection. In this case, the href of the anchor element specifies the URL of the active content, and it isn't necessary to include a panel in the tabset.

If we were to create the markup for a three-tab tabset where all the content is fetched from the server, the markup could be as follows:

```
<div id="tabset">
<ul>
<li><a href="/url/for/panel1">Tab One</a></li>
<li><a href="/url/for/panel2">Tab Two</a></li>
<li><a href="/url/for/panel3">Tab Three</a></li>
</ul>
</div>
```

In this scenario, three elements serving as panels to hold the dynamic content will be automatically created. You can control the id values assigned to these panel elements by placing a title attribute on the anchor. The value of the title, with spaces replaced by underscores, will be the id of the corresponding panel.
You can precreate the panel using this id, and the tab will be correctly hooked up to it, but if you don't, it will be automatically generated. For example, if we were to rewrite the third tab as,

```
<li><a href="/url/for/panel3" title="a third panel">Tab Three</a></li>
```

Command syntax: tabs

tabs(options)
tabs('disable',index)
tabs('enable',index)
tabs('destroy')
tabs('option',optionName,value)
tabs('add',association,label,index)
tabs('remove',index)
tabs('select',index)
tabs('load',index)
tabs('url',index,url)
tabs('length')
tabs('abort')
tabs('rotate',duration,cyclical)
tabs('widget')

Transforms tabset markup (as specified earlier in this section) into a set of UI tabs.
Parameters
options (Object) An object hash of the options to be applied to the tabset, as described in table 11.12.
'disable' (String) Disables one or all tabs. If a zero-based index is provided, only the identified tab is disabled. Otherwise, the entire tabset is disabled.
A backdoor method to disable any set of tabs is to use the data() method to set a data value of disabled.tabs onto the widget element consisting of an array of zero-based indexes of the tabs to be disabled. For example, $('#tabWidget').data('disabled.tabs',[0,3,4]).
'enable' (String) Re-enables a disabled tab or tabset. If a zero-based index is provided, the identified tab is enabled. Otherwise, the entire tabset is enabled.
All tabs can be enabled by using the backdoor trick outlined above, specifying an empty array.
'destroy' (String) Reverts any elements transformed into tab controls to their previous state.
'option' (String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a tab element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.
optionName (String) The name of the option (see table 11.12) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.
index (Number) The zero-based index identifying a tab to be operated upon. Used with disable, enable, remove, select, add, load, and url.

Command syntax: tabs *(continued)*	
'add'	(String) Adds a new tab to the tabset. The index parameter specifies the existing tab before which the new tab will be inserted. If no index is provided, the tab is placed at the end of the tab list.
association	(String) Specifies the association with the panel that will correspond to this tab. This can be an id selector for an existing element to become the panel, or the URL of a server-side resource to create an Ajax tab.
label	(String) The label to assign to the new tab.
'remove'	(String) Removes the indexed tab from the tabset.
'select'	(String) Causes the indexed tab to become the selected tab.
'load'	(String) Forces a reload of the indexed tab, ignoring the cache.
'url'	(String) Changes the association URL for the indexed tab. If the tab isn't an Ajax tab, it becomes one.
url	(String) The URL to a server-side resource that returns a tab's panel content.
'length'	(String) Returns the number of tabs in the first matched tabset in the wrapped set.
'abort'	(String) Aborts any in-progress Ajax tab-loading operations and any running animations.
'rotate'	(String) Sets the tabs to automatically cycle using the specified duration.
duration	(Number) The duration, in milliseconds, between rotations of the tabset. Pass 0 or null to stop an active rotation.
cycle	(Boolean) If true, rotation continues even after a user has selected a tab. Defaults to false.
'widget'	(String) Returns the element serving as the tabs widget, annotated with the ui-tabs class name.
Returns	
	The wrapped set, except for the cases where values are returned as described above.

As might be expected for such a complex widget, there are a fair number of options (see table 11.12).

As usual, we've provided a Tabs Lab to help you sort through the tabs() method options. The Lab can be found in file chapter11/tabs/lab.tabs.html, and it's shown in figure 11.12.

NOTE Because this Lab uses server-side Ajax operations, it must be run from the Tomcat instance we set up for the examples in chapter 8 (note the 8080 port in the URL). Alternatively, you can run this Lab remotely by visiting http://www.bibeault.org/jqia2/chapter11/tabs/lab.tabs.html.

The options available for the tabs() method are shown in table 11.12.

 Table 11.12 Options for the jQuery UI tabs

Option	Description	In Lab?
add	(Function) Specifies a function to be established on the tabset as an event handler for tabsadd events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓

ajaxOptions	(Object) An options hash specifying any additional options to be passed to $.ajax() during any Ajax load operations for the tabset. See the description of the $.ajax() method in chapter 8 for details of these options.	
cache	(Boolean) If true, any content loaded via Ajax will be cached. Otherwise, Ajax content is reloaded. Defaults to false.	✓
collapsible	(Boolean) If true, selecting an already selected tab will cause it to become unselected, resulting in no tab being selected and the pane area collapsing. By default, clicking on an already selected tab has no effect.	✓
cookie	(Object) If provided, specifies that a cookie should be used to remember which tab was last selected and to restore it upon page load. The properties of this object are those expected by the cookie plugin: name, expires (in days), path, domain, and secure. Requires that the cookie plugin (http://plugins.jquery.com/project/cookie) be loaded.	✓
disable	(Function) Specifies a function to be established on the tabset as an event handler for tabsdisable events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓
disabled	(Array) An array containing the zero-based indexes of tabs that will be initially disabled. If the selected option is not specified (defaults to 0), having 0 as index in this array won't disable the first tab, as it will be selected by default.	✓
enable	(Function) Specifies a function to be established on the tabset as an event handler for tabsenable events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓
event	(String) Specifies the event used to select a tab. Most often this is one of click (the default) or mouseover, but events such as mouseoutcan also be specified (even if a bit strange).	✓
fx	(Object) Specifies an object hash to be suitable for use with animate() to be used when animating the tabs. A duration property can be used to specify the duration with any value suitable for the animation method: milliseconds, normal (the default) , slow, or fast. An opacity property can also be specified as a number from 0 to 1.0.	
idPrefix	(String) When no title attribute is present on a tab anchor, specifies the prefix to use when generating a unique id value to assign to the tab panels for dynamic content. If omitted, the prefix ui-tabs-is used.	
load	(Function) Specifies a function to be established on the tabset as an event handler for tabsload events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓

Table 11.12 Options for the jQuery UI tabs *(continued)*

Option	Description	In Lab?
panelTemplate	(String) The HTML template to use when creating tab panels on the fly. This could be the result of an add method or automatic creation for an Ajax tab. By default, the template " " is used.	
remove	(Function) Specifies a function to be established on the tabset as an event handler for tabsremove events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓
select	(Function) Specifies a function to be established on the tabset as an event handler for tabsselect events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓

selected	(Number) The zero-based index of the tab to be initially selected. If omitted, the first tab is selected. The value -1 can be used to cause no tabs to be initially selected.	✓
show	(Function) Specifies a function to be established on the tabset as an event handler for tabsshow events. See the description of the tab events in table 11.13 for details on the information passed to this handler.	✓
spinner	(String) A string of HTML to be displayed in an Ajax tab that's fetching remote content. The default is the string "Loading…". (The embedded HTML entity is the Unicode character for an ellipsis.) In order for the spinner to appear, the content of the tabs anchor element must be a element. For example, Slow	✓
tabTemplate	(String) The HTML template to use when creating new tabs via the add method. If omitted, the default of " #{label}</ span> " is used. Within the template, the tokens #{href} and #{label} are replaced with the values passed to the add method.	

We trust that you've become experienced enough with the various Lab pages presented throughout this book to not need any help working through the basic options in the Tabs Lab. But there are some important nuances we want to make sure you understand around Ajax tabs, so here are a few Lab exercises that you should do after playing around with the basic options:

. *Exercise 1*—Bring up the Lab and, leaving all controls in their default state, click Apply. The Food and Slow tabs are Ajax tabs whose panels aren't loaded until the tabs are selected.

Click the Food tab. This tab is simply loaded from an HTML source and appears instantaneously. But note a *tabsload* event in the console. This indicates that the content was loaded from the server.

Click the Flowers tab and then click the Food tab again. Note how another tabsload event was triggered as the content was loaded again from the server.

Exercise2 —Reset the Lab. Choose the true option for cache, and click Apply.

Repeat the actions of exercise 1 and note how, this time, the Food tab is only loaded on its first selection.

. *Exercise 3*—Reset the Lab and, leaving all controls in their default state, click Apply.

Repeat exercise 1 except click on the Slow tab instead of the Flowers tab. The Slow tab is loaded from a server-side resource that takes about 10 seconds to load. Note how the default spinner value of "Loading ..." is displayed during the lengthy load operation, and how the tabsload event isn't delivered until the content has been received.

. *Exercise 4*—Reset the Lab and, choosing the Image value for the spinner option, click Apply.

Repeat the actions of exercise 3. This supplies the HTML for an <img> element that's displayed in the tab while loading. You can't miss the effect.

Tab events

There are many reasons that we may want to be notified when users are clicking on our tabs. For example, we may want to wait to perform some initialization events on tabbed content until the user actually selects the tab. After all, why do a bunch of work on content that the user may not even look at? The same goes with loaded content. There may be tasks we want to perform after the content has been loaded.

To help us get our hooks into the tabs and tabbed content at the appropriate times, the events shown in table 11.13 are triggered at interesting times during the life of the tabset. Each event handler is passed the event instance as the first parameter, and a custom object as the second, whose properties consist of three elements:

. index—The zero-based index of the tab associated with the event
. tab—A reference to the anchor element for the tab associated with the event
. panel—A reference to the panel element for the tab associated with the event

Table 11.13 Events for jQuery UI tabs

Event	Option	Description
tabsadd	add	Triggered when a new tab is added to the tabset.
tabsdisable	disable	Triggered whenever a tab is disabled.
tabsenable	enable	Triggered whenever a tab is enabled.
tabsload	load	Triggered after the content of an Ajax tab is loaded (even if an error occurs).
tabsremove	remove	Triggered when a tab is removed.
tabsselect	select	Triggered when a tab is clicked upon, becoming selected, unless this callback returns false, in which case the selections is can celed.
tabsshow	show	Triggered when a tabbed panel is shown

As an example, let's say we wanted to add a class name to all image elements in a tabbed panel that loaded via Ajax. We could do that with a single tabsload handler established on the tabset:

```
$('#theTabset').bind('tabsload',function(event,info){  $('img',info.panel).addClass('imageInATab');
});
```

The important points to take away from this small example are

. The info.panel property references the panel affected.
. The panel's content has been loaded by the time the tabsload event is triggered.

Now let's turn our attention to what CSS class names are added to the elements so we can use them as styling hooks.

Styling tabs

When a tabset is created, the following CSS class names are applied to various participating elements:

. ui-tabs—Added to the tabset element
. ui-tabs-nav—Added to the unordered list element housing the tabs
. ui-tabs-selected—Added to the list item representing the selected tab
. ui-tabs-panel—Added to the tabbed panels

Do you think that the tabs are too big in their default rendition? Shrink them down to size with a style rules such as this:

```
ul.ui-tabs-nav { font-size: 0.5em; }
```

Do you want your selected tabs to really stand out? Try this:

```
li.ui-tabs-selected a { background-color: crimson; }
```

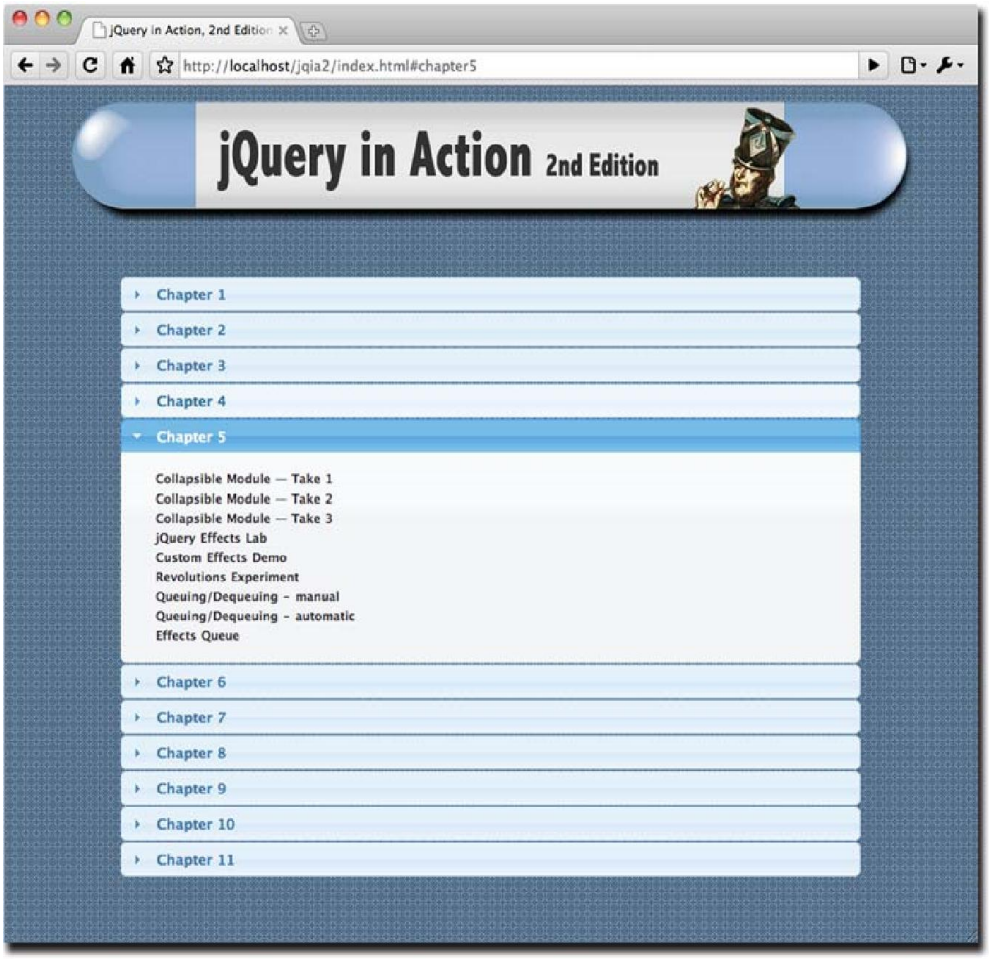
Tabs are a great and ubiquitous widget for organizing panels of related content so that users only see a single panel at a time. But what if they're a bit *too* ubiquitous and you want to achieve the same goal but with a less common look and feel?

An accordion might be just the widget for you.

Accordions

Although the term *accordion* might conjure images of mustached men playing badly delivered tableside serenades, it's actually an apt name for the widget that presents content panels one at a time (just like tabs) in a layout reminiscent of the bellows of the actual instrument.

Rather than having a set of tabs across the top of an area that displays the panels, accordions present choices as a stacked series of horizontal bars, each of whose content is shown between the associated bar and the next. If you've been using the index page for the code examples (index.html in the root folder), you've already seen an accordion in action, as shown in figure 11.13.



Like a tabset, only one panel can be open at a time, and, by default, accordions also adjust the size of the panels so that the widget takes up the same amount of room no matter which panel is open. This makes the accordion a very well-behaved on-page citizen.

Let's take a look at what it takes to create one.

Creating accordion widgets

As with the tabset, the accordion expects a particular HTML construct that it will instrument. Because of the different layout of the accordion, and to make sure things degrade gracefully in the absence of JavaScript, the structure of the source for an accordion is rather different from that for a tabset.

The accordion expects an outer container (to which the accordion() method is applied) that contains pairs consisting of a header and associated content. Rather than using href values to associate content panels to their headers, accordions (by default) expect each header to be followed by its content panel as the next sibling.

A typical construct for an accordion could look like the following:

```
<div id="accordion">

    <h2><a href="#">Header 1</a></h2> <div id="contentPanel_1"> ... content ... </div>

    <h2><a href="#">Header 2</a></h2> <div id="contentPanel_2"> ... content ... </div>

    <h2><a href="#">Header 3</a></h2> <div id="contentPanel_3"> ... content ... </div>
```

Note that the header text continues to be embedded within an anchor—in order to give the user a focusable element—but the href is generally set to # and isn't used to associate the header to its content panel. (There is one option where the anchor's href value is significant, but generally they're just set to #.)

The syntax of the accordion() method is as follows:

Command syntax: accordion

accordion(options)accordion('disable')accordion('enable')accordion('destroy')a
ccordion('option',optionName,value)accordion('activate',index)accordion('widg
et')accordion('resize')

Transforms the accordion source construct (as specified earlier in this section) into an accordion widget.

Parameters

options (Object) An object hash of the options to be applied to the accordion, as described in table 11.14. 'disable' (String) Disables the accordion. 'enable' (String) Re-enables a disabled accordion. 'destroy' (String) Reverts any elements transformed into an accordion widget to their previous state.

'option'(String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be an accordion element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.

optionName(String) The name of the option (see table 11.14) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.

'activate'(String) Activates (opens) the content panel identified by the index parameter.

Command syntax: accordion *(continued)*

index	(Number|Selector|Boolean) A zero-based index identifying the accordion panel to be activated, a selector identifying the panel, or false, which causes all panels to be deactivated if the collapsible option is specified as true.
'widget'	(String) Returns the accordion widget element; the one annotated with the ui-accordion class name.
'resize'	(String) Causes the size of the widget to be recomputed. This should be called whenever something occurs that may cause the widget size to change; for example, resizing its container.

Returns

The wrapped set, except for the cases where values are returned as described above.

The short, but capable, list of options available for the accordion() method is shown in table 11.14.

Follow along in this Lab as you read through the options list in table 11.14.

Table 11.14 Options for the jQuery UI accordions Table 11.14 Options for the jQuery UI accordions *(continued)*

Option	Description	In Lab?
active	(Number|Boolean|Selector|Element|jQuery) Specifies which panel is to be initially open. This can be the zero-based index of the panel, or a means to identify the header element for the panel: an element reference, a selector, or a jQuery wrapped set. If specified as false, no panel is initially opened unless the collapsible options is set to false.	✓
animated	(String|Boolean) The name of the animation to be used when opening and closing accordion panels. One of: slide (the default), bounceslide, or any of the installed easings (if included on the page). If specified as false, no animation is used.	✓
autoHeight	(Boolean) Unless specified as false, all panels are forced to the biggest height needed to accommodate the highest panel, making all panels the same size. Otherwise, panels retain their natural size. Defaults to true.	✓
clearStyle	(Boolean) If true, height and overflow styles are cleared after an animation. The autoHeight option must be set to false for this to apply.	
change	(Function) Specifies a function to be established on the accordion as an event handler for accordionchange events. See the description of the accordion events in table 11.15 for details on the information passed to this handler.	✓
changestart	(Function) Specifies a function to be established on the accordion as an event handler for accordionchangestart events. See the description of the accordion events in table 11.15 for details on the information passed to this handler.	✓
collapsible	(Boolean) If true, clicking on the header for the open accordion panel will cause the panel to close, leaving no panels open. By default, clicks on the open panel's header have no effect.	✓
disabled	(Boolean) If specified and true, the accordion widget is initially disabled.	

Option	Description	In Lab?
event	(String) Specifies the event used to select an accordion header. Most often this is one of click (the default) or mouseover, but events such as mouseout can also be specified (even if a bit strange).	✓
fillSpace	(Boolean) If true, the accordion is sized to completely fill the height of its parent element, overriding any autoHeight option value.	
header	(Selector|jQuery) Specifies a selector or element to override the default pattern for identifying the header elements. The default is "> li > :first-child,> :not(li):even". Use this only if you need to use a source construct for the accordion that doesn't conform to the default pattern.	
icons	(Object) An object that defines the icons to use to the left of the header text for opened and closed panels. The icon to use for closed panels is specified as a property named header, whereas the icon to use for open panels is specified as a property named headerSelected. The values of these properties are strings identifying the icons by class name, as defined earlier for button widgets in section 11.1.3. The defaults are ui-icon-triangle-1-e for header, and ui-icon-triangle-1-s for headerSelected.	✓

navigation (Boolean) If true, the current location (location.href) is used to attempt to match up to the href values of the anchor tags in the accordion headers. This can be used to cause specific accordion panels to be opened when the page is displayed. For example, setting the href values to anchor hashes such as #chapter1 (and so on), will cause the corresponding panel to be opened when the page is displayed if the URL (or bookmark) is suffixed with the same hash value. The index.html page for the code examples uses this technique. Try it out! Visit the page by specifying index.html#chapter3 as part of the URL.

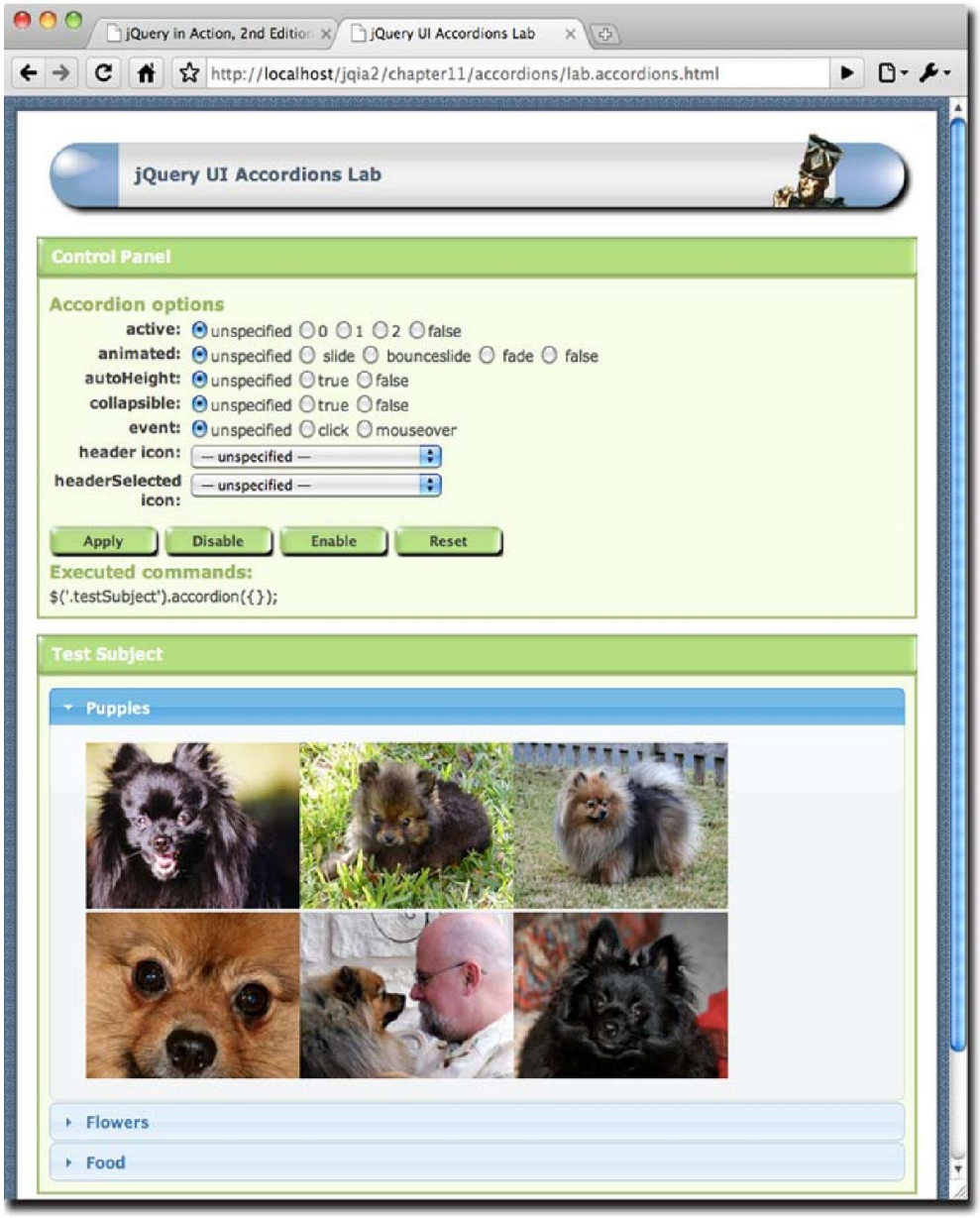
navigationFilter (Function) Overrides the default navigation filter used when navigation is true. You can use this function to change the behavior described in the navigation option description to any of your own choosing. This callback will be invoked with no parameters, and the anchor tag for a header is set as the function context. Return true to indicate that a navigation match has occurred.

We've provided the Accordions Lab, in file chapter11/accordions/lab/accordions.html, to demonstrate many of the options. It's shown in figure 11.14.

After you've run through the basic options and tried out things in the Accordions Lab, here are a couple of exercises we want to make sure you don't miss:

. *Exercise 1*—Load the Lab and, leaving all settings at their default, click Apply. Select various headers in any order and note how, as the panels open and close, the accordion itself never changes size.

. *Exercise 2*—Reset the Lab, choose true for autoHeight, and click Apply. Run through the actions of exercise 1, noticing that, this time, when the Flowers panel is opened, the height of the accordion shrinks to fit the smaller content of the Flowers panel.



Now we're ready to tackle the events that are triggered while an accordion is being manipulated.

Accordion events

Accordions trigger only two event types when the user is opening and closing panels, as described in table 11.15.

Each of the handlers is passed the usual event instance and custom object. The properties of the custom object are the same for both events and consist of the following: options—The options passed to the accordion() method when the widget was created. oldHeader—A jQuery wrapped set containing the header element of the previously open panel. This may be empty if no panel was opened.

. newHeader—A jQuery wrapped set containing the header element of the panel being opened. This may be empty for collapsible accordions when all panels are being closed.

oldContent—A jQuery wrapped set containing a reference to the previously open panel.

. newContent—A jQuery wrapped set containing a reference to the panel being opened.

Table 11.15. lists the events generated for accordion widgets.

Table 11.15 Events for jQuery UI accordions

Event	Option	Description
accordionchangestart accordionchange	changestart change	Triggered when the accordion is about to change. Triggered when the accordion has been changed, after the duration of any animation used to change the display.

That's a pretty sparse list of events, and it does offer some challenges. For example, it's disappointing that we get no notification when the initial panel (if any) is opened. We'll see how that makes things a tad harder for us when we try to use these events to instrument the accordion. But before we tackle an example of using these events to add some functionality to our widget, let's examine the CSS class names that jQuery UI adds to the elements that compose the accordion.

Styling classes for accordions

As with tabs, jQuery UI adds a number of CSS class names to the elements that go into making an accordion. We can use not only use them as styling hooks, but to find the elements using jQuery selectors. We saw an example of that in the previous section when we learned how to locate the panels involved in the accordion events.

These are the class names that are applied to the accordion elements:

. ui-accordion—Added to the outer container for the accordion (the element upon which accordion() was called).

. ui-accordion-header—Added to all header elements that become the click-able elements.

. ui-accordion-content—Added to all panel elements.

. ui-accordion-content-active—Assigned to the currently open panel element, if any.

. ui-state-active—Assigned to the header for the currently open panel, if any. Note that this is one of the generic jQuery UI class names shared across multiple widgets.

Using these class names, we can restyle accordion elements as we like, much like we did for tabs. Try your hand at changing the style of the elements: the header text, for example, or maybe the border surrounding the panels.

Let's also see how knowing these class names helps us to add functionality to an accordion widget.

Loading accordion panels using Ajax

One feature that the accordion widget lacks, present in its tabs widget kinfolk, is the innate ability to load content via Ajax. Not wanting our accordions to suffer from an inferiority complex, let's see how we can easily add that ability using the knowledge that we have at hand.

Tabs specify the location of remote content via the href of the anchor tags within them. Accordions,

on the other hand, ignore the href of anchor tags in their header unless the navigation option is being used. Knowing this, we'll safely use it to specify the location of any remote content to be loaded for the panel.

This is a good decision because it's consistent with the way that tabs work (consistency is a good thing), and it means we don't have to needlessly introduce custom options or attributes to record the location. We'll leave the anchor href of "normal" panels at #.

We want to load the panel content whenever the panel is about to open, so we bind an accordionchangestart event to the accordion(s) on our page with this code:

```
$('.ui-accordion').bind('accordionchangestart',function(event,info){  if (info.newContent.length == 0) return;  var href =
    $('a',info.newHeader).attr('href');  if (href.charAt(0) != '#') {
       info.newContent.load(href);
    $('a',info.newHeader).attr('href','#'); }});
```

In this handler we first locate the opening panel by using the reference provided in info.newContent. If there's none (which can happen for collapsible accordions), we simply return.

Then we locate the anchor within the activating header by finding the <a> element within the context of the reference provided by info.newHeader, and grab its href attribute. If it doesn't start with #, we assume it's a remote URL for the panel content.

To load the remote content, we employ the handy load() method, and then change the href of the anchor to #. This last action prevents us from fetching the content again next time the panel is opened. (To force a load every time, simply remove the href assignment.)

When using this handler, we might want to turn autoHeight off if not knowing the size of the largest panel in advance creates a problem. A working example of this approach can be found at chapter11/accordions/ajax/ajax-accordion.html.

As usual, there are always different vectors of approach. Try the following exercise.

. *Exercise 1*—If we wanted to avoid using the href value so that we could use the navigation option, how would you rewrite the example to use custom attributes (or any other tactic of your choosing)?

Accordions give us an alternative to tabbed panels when we want to serially present related content to the user. Now let's wrap up our examination of the widgets, by looking at another widget that lets us present content dynamically.

Dialog boxes

As a concept, dialog boxes need no introduction. A staple of desktop application design since the inception of the GUI, dialog boxes, whether modeless or modal, are a common means of eliciting information from the user, or delivering information to the user.

In web interfaces, however, they haven't existed as an innate concept except for the built-in JavaScript alert prompt and confirm tools. Deemed inadequate for a variety of reasons—not the least of which is their inability to be styled to conform to the theme of a site—these tools are often ignored except as debugging aids.

Internet Explorer introduced the concept of a web-based dialog box, but it failed to impress the standards community and remains a proprietary solution.

For years, web developers used the window.open() method to create new windows that stood in for dialog boxes. Although fraught with issues, this approach was adequate as a solution for modeless dialog boxes, but truly modal dialog boxes were out of reach.

As JavaScript, browsers, DOM manipulations, and developers themselves have become more capable, it's become possible to use these basic tools to create in-page elements that "float" over the rest of the display—even locking out input in a modal fashion—which better approximates the semantics of

modeless and modal dialog boxes.

So although dialog boxes as a concept still don't actually exist in web interfaces, we can do a darned good job of making it seem like they do. Let's see what jQuery UI provides for us in this area.

Creating dialog boxes

Although the idea of in-page dialog boxes seems simple—just remove some content from the page flow, float it with a high z-index, and put some "chrome" around it—there are lots of details to take into account. Luckily, jQuery UI is going to handle that, allowing us to create modeless and modal dialog boxes, with advanced features such as the ability to be resized and repositioned, with ease.

NOTE The term "chrome" when applied to dialog boxes denotes the frame and widgets that contain the dialog box and allow it to be manipulated. This can include features such as resize borders, title bar, and the omnipresent little "x" icon that closes the dialog box.

Unlike the rather stringent requirements for the tabs and accordion widgets, just about any element can become the body of a dialog box, though a element containing the content that's to become the dialog box's body is most often used.

To create a dialog box, the content to become the body is selected into a wrapped set, and the dialog() method is applied. The dialog() method has the following syntax:

Command syntax: dialog

dialog(options)dialog('disable')dialog('enable')dialog('destroy')dialog('option',optionName,value)dialog('open')dialog('close')dialog('isOpen')dialog('moveToTop')dialog('widget')

Transforms the elements in the wrapped set into a dialog box by removing them from the document flow and wrapping them in "chrome." Note that creating a dialog box also causes it to automatically be opened unless this is disabled by setting the autoOpen option to false.

Parameters

options (Object) An object hash of the options to be applied to the dialog box, as described in table 11.16.'disable' (String) Disables the dialog box.'enable' (String) Re-enables a disabled dialog box.

'destroy'(String) Destroys the dialog box. Once destroyed, the dialog box can't bereopened. Note that destroying a dialog box doesn't cause the contained elements to be restored to the normal document flow.

'option'(String) Allows option values to be set on all elements of the wrapped set, or to be retrieved from the first element of the wrapped set (which should be a dialog box element), based upon the remaining parameters. If specified, at least the optionName parameter must also be provided.

optionName(String) The name of the option (see table 11.16) whose value is to be set or returned. If a value parameter is provided, that value becomes the option's value. If no value parameter is provided, the named option's value is returned.

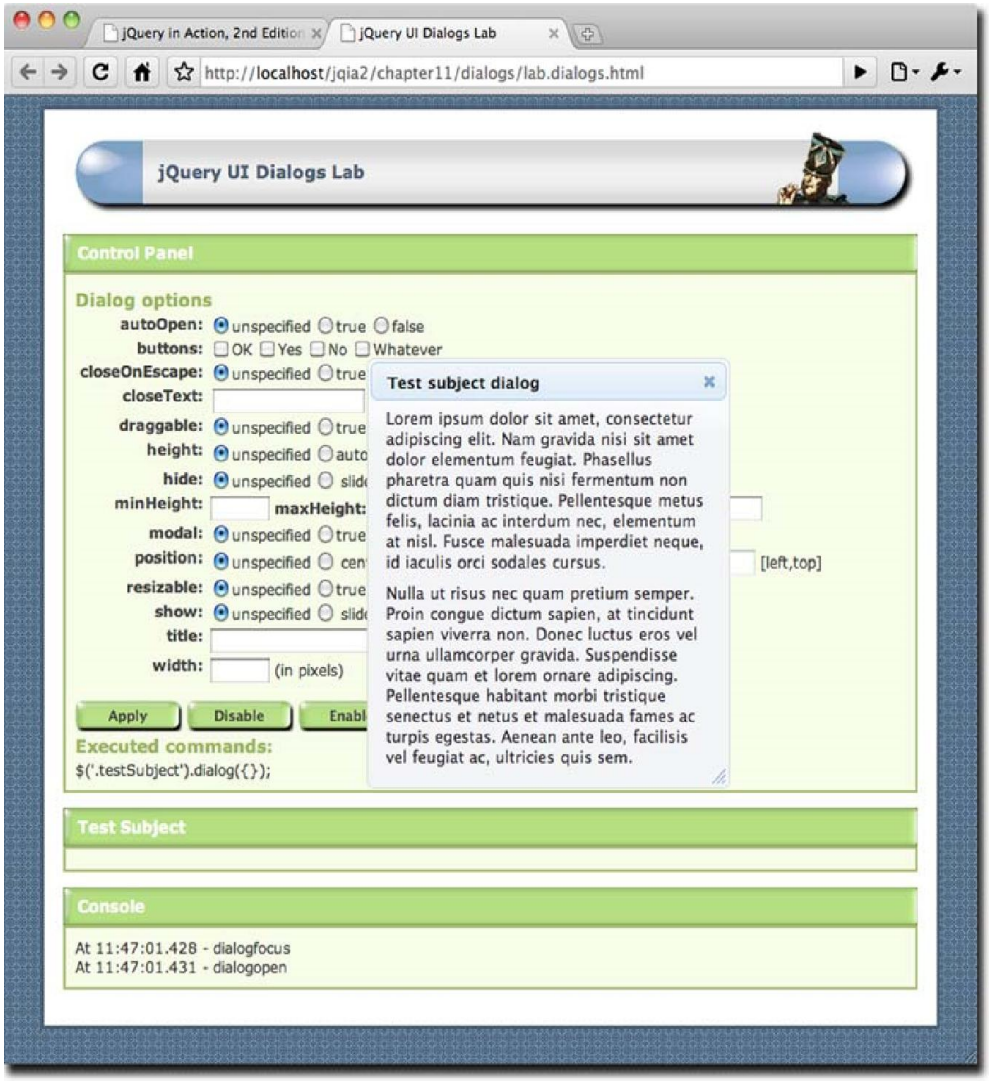
'open' (String) Opens a closed dialog box.'close' (String) Closes an open dialog box. The dialog box can be reopened at any time with the open method. 'isOpen' (String) Returns true if the dialog box is open; false otherwise. 'moveToTop' (String) If multiple dialog boxes exist, moves the dialog box to the top of the stack of dialog boxes.

Command syntax: dialog *(continued)*

'widget'(String) Returns the dialog box's widget element; the element annotated with the ui-dialog class name.

Returns
The wrapped set, except for the cases where values are returned as described above.

It's important to understand the difference between creating a dialog box and opening one. Once a dialog box is created, it doesn't need to be created again to be reopened after closing. Unless disabled, a dialog box is automatically opened upon creation, but to reopen a dialog box that has been closed, we call dialog('open') rather than calling the dialog() method with options again.



As usual, a Dialogs Lab has been made available in file chapter11/dialogs/lab.dialogs.html, shown in figure 11.15, so you can try out the dialog() method options. Follow along in this Lab as you

read through the options list in table 11.16.

Table 11.16 Options for the jQuery UI dialogs Table 11.16 Options for the jQuery UI dialogs *(continued)*

Option	Description	In Lab?
autoOpen	(Boolean) Unless set to false, the dialog box is opened upon creation. When false, the dialog box will be opened upon a call to dialog('open').	✓
beforeClose	(Function) Specifies a function to be established on the dialog box as an event handler for dialogbeforeClose events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	✓
buttons	(Object) Specifies any buttons to place at the bottom of the dialog box. Each property in the object serves as the caption for the button, and the value must be a callback function to be invoked when the button is clicked. This handler is invoked with a function context of the dialog box element, and is passed the event instance with the button set as the target property. If omitted, no buttons are created for the dialog box. The function context is suitable for use with the dialog() method. For example, within a Cancel button, the following could be used to close the dialog box: $(this).dialog('close');	✓
close	(Function) Specifies a function to be established on the dialog box as an event handler for dialogclose events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	✓
closeOnEscape	(Boolean) Unless set to false, the dialog box will be closed when the user presses the Escape key while the dialog box has focus.	✓
closeText	(String) Text to replace the default of Close for the close button.	✓
dialogClass	(String) Specifies a space-delimited string of CSS class names to be applied to the dialog box element in addition to the class names that jQuery UI will add. If omitted, no extra class names are added.	
drag	(Function) Specifies a function to be established on the dialog box as an event handler for drag events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	✓
dragstart	(Function) Specifies a function to be established on the dialog box as an event handler for dragStart events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
dragstop	(Function) Specifies a function to be established on the dialog box as an event handler for dragStop events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
draggable	(Boolean) Unless set to false, the dialog box is draggable by clicking and dragging its title bar.	✓
focus	(Function) Specifies a function to be established on the dialog box as an event handler for dialogfocus events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	

Table 11.16 Options for the jQuery UI dialogs *(continued)*

Option	Description	In Lab?
`height`	(Number|String) The height of the dialog box in pixels, or the string `"auto"` (the default), which allows the dialog box to determine its height based upon its contents.	✓
`hide`	(String) The effect to be used when the dialog box is closed (as we discussed in chapter 9). By default, none.	✓
`maxHeight`	(Number) The maximum height, in pixels, to which the dialog box can be resized.	✓
`maxWidth`	(Number) The maximum width, in pixels, to which the dialog box can be resized.	✓
`minHeight`	(Number) The minimum height, in pixels, to which the dialog box can be resized. Defaults to 150.	✓
`minWidth`	(Number) The minimum width, in pixels, to which the dialog box can be resized. Defaults to 150.	✓
`modal`	(Boolean) If `true`, a semi-transparent "curtain" is created behind the dialog box covering the remainder of the window content, preventing any user interaction. If omitted, the dialog box is modeless.	✓
`open`	(Function) Specifies a function to be established on the dialog box as an event handler for `dialogopen` events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
`position`	(String|Array) Specifies the initial position of the dialog box. Can be one of the predefined positions: `center` (the default), `left`, `right`, `top`, or `bottom`. Can also be a 2-element array with the left and top values (in pixels) as `[left,top]`, or text positions such as `['right','top']`.	✓
`resize`	(Function) Specifies a function to be established on the dialog box as an event handler for resize events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
`resizable`	(Boolean) Unless specified as `false`, the dialog box is resizable in all directions.	✓
`resizeStart`	(Function) Specifies a function to be established on the dialog box as an event handler for `resizeStart` events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
`resizeStop`	(Function) Specifies a function to be established on the dialog box as an event handler for `resizeStop` events. See the description of the dialog events in table 11.17 for details on the information passed to this handler.	
`show`	(String) The effect to be used when the dialog box is being opened. By default, no effect is used.	✓
`stack`	(Boolean) Unless specified as `false`, the dialog box will move to the top of any other dialog boxes when it gains focus.	
`title`	(String) Specifies the text to appear in the title bar of the dialog box chrome. By default, the `title` attribute of the dialog box element will be used as the title.	✓

Option	Description	In Lab?
width (Number)	The width of the dialog box in pixels. If omitted, a default of 300 pixels is used.	✓
zIndex (Number)	The initial z-index for the dialog box, overriding the default value of 1000.	

Most of these options are easy to see in action using the Dialogs Lab, but make sure you run through the differences between modal and modeless dialog boxes.

In the console of the Lab, the various events that are triggered (as the dialog box is interacted with) are displayed in the order that they're received. Let's examine the possible events.

11.8.2 Dialog events

As the user manipulates the dialog boxes we create, various custom events are triggered that let us get our hooks into the page. This gives us the opportunity to perform actions at pertinent times during the life of the dialog box, or even to affect the operation of the dialog box.

The events triggered during dialog box interactions are shown in table 11.17. Each of these handlers is passed the event instance and a custom object. The function context, as well as the event target, is set to the dialog box element.

The custom object passed to the handler depends upon the event type:

. For the drag, dragStart, and dragStop events, the custom object contains properties offset and position, which in turn contain left and top properties that identify the position of the dialog box relative to the page or its offset parent respectively.

. For the resize, resizeStart, and resizeStop events, the custom object contains the properties originalPosition, originalSize, position, and size. The position properties are objects that contain the expected left and top properties, while the size properties contain height and width properties.

. For all other event types, the custom object has no properties.

Table 11.17 Events for jQuery UI dialogs Table 11.17 Events for jQuery UI dialogs *(continued)*

Event	Option	Description
dialogbeforeClose dialogclose	beforeClose close	Triggered when the dialog box is about to close. Returning false prevents the dialog box from closing—handy for dialog boxes with forms that fail validation. Triggered after a dialog box has closed.

Event	Option	Description

drag dragStart dragStop dialogfocus dialogopen resize resizeStart resizeStop	drag dragStart dragStop focus open resize resizeStart resizeStop	Triggered repeatedly as a dialog box is moved about during a drag. Triggered when a repositioning of the dialog box commences by dragging its title bar. Triggered when a drag operation terminates. Triggered when the dialog box gains focus. Triggered when the dialog box is opened. Triggered repeatedly as a dialog box is resized. Triggered when a resize of the dialog box commences. Triggered when a resize of the dialog box terminates.

Before we can see a few clever uses of these events, let's examine the class names that jQuery places on the elements that participate in the creation of our dialog boxes.

11.8.3 Dialog box class names

As with the other widgets, jQuery UI marks up the elements that go into the structure of the dialog box widget with class names that help us to find the elements, as well as to style them via CSS.

In the case of dialog boxes, the added class names are as follows:

. ui-dialog—Added to the element created to contain the entire widget, including the content and the chrome.

. ui-dialog-titlebar—Added to the element created to house the title and close icon.

. ui-dialog-title—Added to the element contained within the title bar to wrap the title text.

. ui-dialog-titlebar-close—Added to the <a> tag used to encompass the 'x' icon within the title bar.

. ui-dialog-content—Added to the dialog box content element (the element wrapped during the call to dialog()).

It's important to remember that the element passed to the event handlers is the dialog box content element (the one marked with ui-dialog-content), not the generated outer container created to house the widget.

Now let's look at a few ways to specify content that's not already on the page.

Some dialog box tricks

Generally, dialog boxes are created from elements that are included in the page markup. jQuery UI takes that content, removes it from the DOM, creates elements that serve as the dialog box chrome, and sets the original elements as the content of the chrome.

But what if we wanted to load the content dynamically upon dialogopen via Ajax? That's actually surprisingly easy with code such as this:

```
$('<div>').dialog({  open: function(){ $(this).load('/url/to/resource'); },  title: 'A dynamically loaded dialog'
});
```

In this code, we create a new element on the fly, and turn it into a dialog box just as if it were an

existing element. The options specify its title, and a callback for dialogopen events that loads the content element (set as the function context) using the load() method.

In the scenarios we've seen so far, regardless of whether the content already existed on the page or was loaded via Ajax, the content exists within the DOM of the current page. What if we want the dialog box body to be its own page?

Although it's convenient to have the dialog box content be part of the same DOM as its parent, if the dialog box content and the page content need to interact in any way, we might want the dialog box content to be a separate page unto itself. The most common reason may be because the content needs its own styles and scripts that we don't want to include in every parent page in which we plan to use the dialog box.

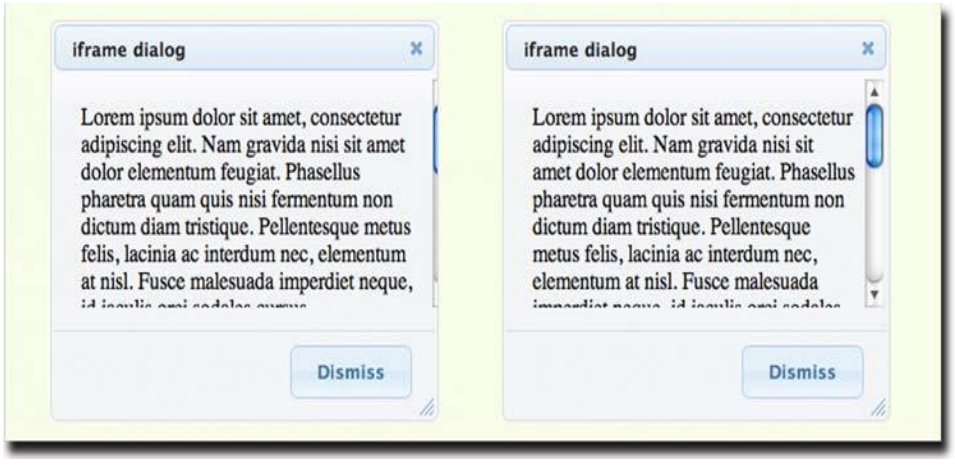
How could we accomplish this? Is there support in HTML for using a separate page as a part of another page? Of course ... the <iframe> element! Consider this:

```
$('<iframe src="content.html" id="testDialog">').dialog({  title: 'iframe dialog',  buttons: {
        Dismiss: function(){ $(this).dialog('close'); }
  }
 });
```

Here we dynamically create an <iframe> element, specifying its source and an id, and make it into a dialog box. The options we pass to the dialog() method give the dialog box its title and a Dismiss button that closes the dialog box. Is this awesome or what?

But our self-admiration is short-lived when we display the dialog box and see a problem. The scrollbar for the <iframe> is clipped by the dialog chrome, as shown in the left half of figure 11.16. What we want, of course, is for the dialog box to appear as shown in the right half of the figure.

Because the <iframe> appears a bit too wide, we could attempt to narrow it with a CSS rule, but to our chagrin, we find that doesn't work. A little digging reveals that a CSS style of width: auto is placed right on the <iframe> element by the dialog() method, defeating any attempt to style the <iframe> indirectly.



But that's OK. We'll just use a bigger sledgehammer. Let's add the following option to the dialog() call:

```
open: function(){  $(this).css('width','95%');
}
```

This overrides the style placed on the <iframe> when the dialog box is opened.

Bear in mind that this approach isn't without its pitfalls. For example, any buttons

are created in the parent page, and interaction between the buttons and the page loaded into the <iframe> will need to communicate across the two windows.

The source for this example can be found in file chapter11/dialogs/iframe.dialog.html.

Summary

Wow. This was a long chapter, but we learned a great deal from it.

We saw how jQuery UI builds upon the interactions and effects that it provides, and which we explored in the previous chapters, to allow us to create various widgets that help us present intuitive and easy-to-use interfaces to our users.

We learned about the button widget that augments the look and feel of conventional HTML buttons so that they play well in the jQuery UI sandbox.

Widgets that allow our users to enter data types that have traditionally been fraught with problems, namely numeric and date data, are provided in the guise of sliders and datepickers. Autocomplete widgets round out the data entry widgets, letting users quickly filter through large sets of data.

Progress bars give us the ability to communicate completion percentage status to our users in a graphical, easy-to-understand display. And finally, we saw three widgets that let us organize our content in varying fashions: tabs, the accordion, and the dialog box.

Added to our toolbox, these widgets give us a wider range of possibilities for our interfaces. But that's just the official set of widgets provided by jQuery UI. As we've seen firsthand, jQuery is designed to extend easily, and the jQuery community hasn't been sitting on its hands. Hundreds, if not many thousands, of other plugin controls exist, just waiting for us to discover them. A good place to start is http://plugins.jquery.com/.

The end?

Hardly! Even though we've presented the entire API for jQuery and jQuery UI within the confines of this book, it would have been impossible to show you all the many ways that these broad APIs can be used on our pages. The examples we presented were chosen specifically to lead you down the path of discovering how you can use jQuery to solve the problems that you encounter on a day-to-day basis on your web application pages.

jQuery is a living project. Astoundingly so! Heck, it was quite a chore for your authors to keep up with the rapid developments in the libraries over the course of writing this book. The core library is constantly evolving into a more useful resource, and more and more plugins are appearing on practically a daily basis. And the pace of development for jQuery UI is practically exhausting.

We urge you to keep track of the developments in the jQuery community and sincerely hope that this book has been a great help in starting you on the path to writing better web applications in less time and with less code than you might have ever believed possible.

We wish you health and happiness, and may all your bugs be easily solvable!