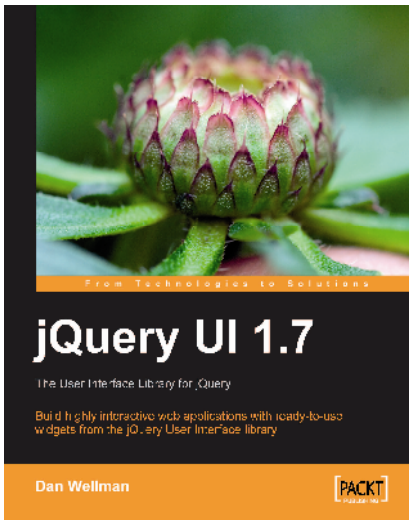
jQuery UI 1.7

The User Interface Library for jQuery

Dan Wellman



Chapter No. 3

"Tabs"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.3 "Tabs"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Dan Wellman lives with his wife and three children in his home town of Southampton on the south coast of England. By day his mild-mannered alter-ego works for a small yet accomplished e-commerce production agency. By night he battles the forces of darkness and fights for truth, justice, and less intrusive JavaScript.

He has been writing computer-related articles, tutorials, and reviews for around five years and is rarely very far from a keyboard of some description. This is his third book.

I'd like to thank the Packt editorial team, all of the technical reviewers and the jQuery UI team, without whom this book would not have been possible. Special thanks go to Jörn Zaefferer who provided essential feedback and was always happy to answer my late-night, niggling questions.

Thanks also to my fantastic friends, in no particular order; Steve Bishop, Eamon O' Donoghue, James Zabiela, Andrew Herman, Aaron Matheson, Dan Goodall, Mike Woodford, Mike Newth, John Adams, Jon Field and Vicky Hammond and all the rest of the guys and girls.

jQuery UI 1.7

The User Interface Library for jQuery

Modern web application user interface design requires rapid development and proven results. jQuery UI, a trusted suite of official plugins for the jQuery JavaScript library, gives you a solid platform on which to build rich and engaging interfaces with maximum compatibility, stability, and a minimum of time and effort.

jQuery UI has a series of ready-made, great looking user interface widgets, and a comprehensive set of core interaction helpers designed to be implemented in a consistent and developer-friendly way. With all this, the amount of code that you need to write personally to take a project from conception to completion is drastically reduced.

Specially revised for version 1.7 of jQuery, this book has been written to maximize your experience with the library by breaking down each component and walking you through examples that progressively build upon your knowledge, taking you from beginner to advanced usage in a series of easy-to-follow steps.

In this book, you'll learn how each component can be initialized in a basic default implementation and then see how easy it is to customize its appearance and configure its behavior to tailor it to the requirements of your application. You'll look at the configuration options and the methods exposed by each component's API to see how these can be used to bring out the best of the library.

Events play a key role in any modern web application if it is to meet the expected minimum requirements of interactivity and responsiveness, and each chapter will show you the custom events fi red by the component covered and how these events can be intercepted and acted upon.

What This Book Covers

Chapter 1, *Introducing jQuery UI*, gives a general overview and introduction to jQuery UI. You find out exactly what the library is, where it can be downloaded from, and where resources for it can be found. We look at the freedom the license gives you to use the library and how the API has been simplified to give the components a consistent and easy to use programming model.

In Chapter 2, *The CSS Framework*, we look in details at the extensive CSS framework that provides a rich environment for integrated theming via ThemeRoller, or allows developers to easily supply their own custom themes or skins.

In Chapter 3, *Tabs*, we look at the tabs component; a simple but effective means of presenting structured content in an engaging and interactive widget.

Chapter 4, *The Accordion Widget*, looks at the accordion widget, another component dedicated to the effective display of content. Highly engaging and interactive, the accordion makes a valuable addition to any web page and its API is exposed in full to show exactly how it can be used.

In Chapter 5, *The Dialog*, we focus on the dialog widget. The dialog behaves in the same way as a standard browser alert, but it does so in a much less intrusive and more visitor-friendly manner. We look at how it can be configured and controlled to provide maximum benefit and appeal.

Chapter 6, *Slider*, looks at the slider widget that provides a less commonly used, but no less valuable user interface tool for collecting input from your visitors. We look closely at its API throughout this chapter to see the variety of ways that in which it can be implemented.

In Chapter 7, *Datepicker*, we look at the datepicker. This component packs a huge amount of functionality into an attractive and highly usable tool, allowing your visitors to effortlessly select dates. We look at the wide range of configurations that its API makes possible as well as seeing how easy common tasks such as skinning and localization are made.

In Chapter 8, *Progressbar*, we look at the new progressbar widget; examining its compact API and seeing a number of ways in which it can be put to good use in our web applications.

In Chapter 9, *Drag and Drop*, we begin looking at the low-level interaction helpers, tackling first the related drag-and-droppable components. We look at how they can be implemented individually and how they can be used together for maximum effect.

Chapter 10, *Resizing*, looks at the resizing component and how it is used with the dialog widget from earlier in the book. We see how it can be applied to any element on the page to allow it be resized in a smooth and attractive way.

In Chapter 11, *Selecting*, we look at the selectable component, which allows us add behavior to elements on the page and allow them be selected individually or as a group. We see that this is one component that really brings the desktop and the browser together as application platforms.

Chapter 12, *Sorting*, looks at the final interaction helper—the sortable component. This is an especially effective component that allows you to create lists on a page that can be reordered by dragging items to a new position on the list. This is another component that can really help you to add a high level of professionalism and interactivity to your site with a minimum of effort.

Chapter 13, *UI Effects*, is dedicated solely to the special effects that are included with the library. We look at an array of different effects that allow you to show, hide, move, and jiggle elements in a variety of attractive and appealing animations. There is no 'fun with' section at the end of this chapter; the whole chapter is a 'fun with' section.

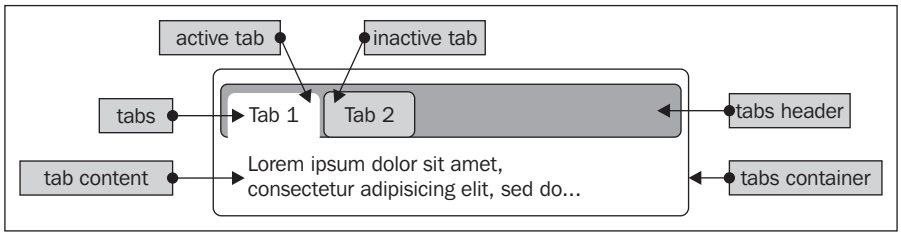
3
Tabs

Now that we've been formally introduced to the jQuery UI library and the CSS framework we can move on to begin looking at the components included in the library. Over the next six chapters, we'll be looking at the widgets. These are a set of visually engaging, highly configurable user interface widgets built on top of the foundation provided by the low-level interaction helpers.

The UI tabs widget is used to toggle visibility across a set of different elements, each element containing content that can be accessed by clicking on its heading which appears as an individual tab.

The tabs are structured so that they line up next to each other, whereas the content sections are layered on top of each other, with only the top one visible. Clicking a tab will highlight the tab and bring its associated content panel to the top of the stack. Only one content panel can be open at a time.

The following screenshot shows the different components of a set of UI tabs:



In this chapter, we will look at the following topics:

- The default implementation of the widget
- How the CSS framework targets tab widgets
- How to apply custom styles to a set of tabs

- Configuring tabs using their options
- Built-in transition effects for content panel changes
- Controlling tabs using their methods
- Custom events defined by tabs
- AJAX tabs

A basic tab implementation

The structure of the underlying HTML elements, on which tabs are based, is rigid and widgets require a certain number of elements for them to work.

The tabs must be created from a list element (ordered or unordered) and each list item must contain an `<a>` element. Each link will need to have a corresponding element with a specified `id` that it is associated with the link's `href` attribute. We'll clarify the exact structure of these elements after our first example.

In a new file in your text editor, create the following page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css"
      href="development-bundle/themes/smoothness/ui.all.css">
    <meta http-equiv="Content-Type"
      content="text/html; charset=utf-8">
    <title>jQuery UI Tabs Example 1</title>
  </head>
  <body>
    <div id="myTabs">
      <ul>
        <li><a href="#a">Tab 1</a></li>
        <li><a href="#b">Tab 2</a></li>
      </ul>
      <div id="a">This is the content panel linked to the first tab,
        it is shown by default.</div>
      <div id="b">This content is linked to the second tab and will
        be shown when its tab is clicked.</div>
    </div>
    <script type="text/javascript"
      src="development-bundle/jquery-1.3.2.js"></script>
    <script type="text/javascript"
      src="development-bundle/ui/ui.core.js"></script>
    <script type="text/javascript"
```

```
    src="development-bundle/ui/ui.tabs.js"></script>
  <script type="text/javascript">
    $(function(){

      $("#myTabs").tabs();
    });
  </script>
 </body>
</html>
```

Save the code as `tabs1.html` in your `jqueryui` working folder. Let's review what was used. The following script and CSS resources are needed for the default tab widget instantiation:

- `ui.all.css`
- `jquery-1.3.2.js`
- `ui.core.js`
- `ui.tabs.js`

A tab widget consists of several standard HTML elements arranged in a specific manner (these can be either hardcoded into the page, or added dynamically, or can be a mixture of both depending on the requirements of your implementation).

- An outer container element, which the `tabs` method is called on
- A list element (`<ul>` or `<ol>`)
- An `<a>` element for each tab
- An element for the content of each tab

The first two elements within the outer container make the clickable tab headings, which are used to show the content section that is associated with the tab. Each tab should include a list item containing the link.

The `href` attribute of the link should be set as a fragment identifier, prefixed with `#`. It should match the `id` attribute of the `element` that forms the content section, with which it is associated. The content sections of each tab are created by the `<div>` elements. The `id` attribute is required and will be targeted by its corresponding `<a>` element. We've used `<div>` elements in this example as the content panels for each tab, but other elements, such as `<span>` elements can also be used.

The elements discussed so far, along with their required attributes, are the minimum that are required from the underlying markup.

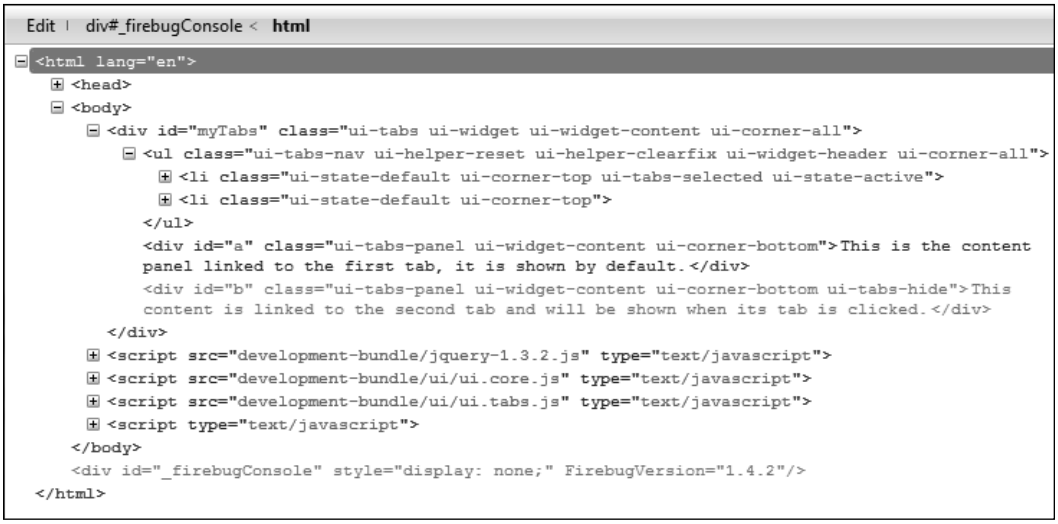
We link to several `<script>` resources from the library at the bottom of the `<body>` before its closing tag. Loading scripts last, after stylesheets and page elements is a proven technique for improving performance. After linking first to jQuery, we link to the `ui.core.js` file that is required by all components (except the effects, which have their own core file). We then link to the component's source file that in this case is `ui.tabs.js`.

After the three required script files from the library, we can turn to our custom `<script>` element in which we add the code that creates the tabs. We simply use the `$(function(){});` shortcut to execute our code when the document is ready. We then call the `tabs()` widget method on the jQuery object, representing our tabs container element (the `<ul>` with an `id` of `myTabs`).

When we run this file in a browser, we should see the tabs as they appeared in the first screenshot of this chapter (without the annotations of course).

Tab CSS framework classes

Using Firebug for Firefox (or another generic DOM explorer) we can see that a variety of class names are added to the different underlying HTML elements that the tabs widget is created from, as shown in the following screenshot:



Let's review these briefly. To the outer container `<div>` the following class names are added:

Class name	Purpose
`ui-tabs`	Allows tab-specific structural CSS to be applied.
`ui-widget`	Sets generic font styles that are inherited by nested elements.
`ui-widget-content`	Provides theme-specific styles.
`ui-corner-all`	Applies rounded corners to container.

The first element within the container is the `<ul>` element. This element receives the following class names:

Class name	Purpose
`ui-tabs-nav`	Allows tab-specific structural CSS to be applied.
`ui-helper-reset`	Neutralizes browser-specific styles applied to `<ul>` elements.
`ui-helper-clearfix`	Applies the clear-fix as this element has children that are floated.
`ui-widget-header`	Provides theme-specific styles.
`ui-corner-all`	Applies rounded corners.

The individual `<li>`elements are given the following class names:

Class name	Purpose
`ui-state-default`	Applies theme-specific styles.
`ui-corner-top`	Applies rounded corners to the top edges of the elements.
`ui-tabs-selected`	This is only applied to the active tab. On page load of the default implementation this will be the first tab. Selecting another tab will remove this class from the currently selected tab and apply it to the new tab.
`ui-state-active`	Applies theme-specific styles to the currently selected tab. This class name will be added to the tab that is currently selected, just like the previous class name. The reason there are two class names is that `ui-tabs-selected` provides the functional CSS, while `ui-state-active` provides the visual, decorative styles.

The `<a>` elements within each `<li>` are not given any class names, but they still have both structural and theme-specific styles applied to them by the framework.

Finally, the elements that hold each tab's content are given the following class names:

Class name	Purpose
`ui-tabs-panel`	Applies structural CSS to the content panels.
`ui-widget-content`	Applies theme-specific styles.
`ui-corner-bottom`	Applied rounded corners to the bottom edges of the content panels.

All of these classes are added to the underlying elements automatically by the library, we don't need to manually add them when coding the page.

As these tables illustrate, the CSS framework supplies the complete set of both structural CSS styles that control how the tabs function and theme-specific styles that control how the tabs appear, but not how they function. We can easily see which selectors we'll need to override if we wish to tweak the appearance of the widget, which is what we'll do in the following section.

Applying a custom theme to the tabs

In the next example, we can see how to change the tabs' basic appearance. We can override any rules used purely for display purposes with our own style rules for quick and easy customization without changing the rules related to the tab functionality or structure.

In a new file in your text editor, create the following very small stylesheet.

```
#myTabs {
  border:1px solid #636363; width:400px;
  background:#c2c2c2 none; padding:5px;
}
.ui-widget-header {
  border:0; background:#c2c2c2 none; font-family:Georgia;
}
#myTabs .ui-widget-content {
  border:1px solid #aaaaaa; background:#ffffff none;
  font-size:80%;
}
.ui-state-default, .ui-widget-content .ui-state-default {
  background:#a2a2a2 none; border:1px solid #636363;
}
.ui-state-active, .ui-widget-content .ui-state-active {
  background:#ffffff none; border:1px solid #aaaaaa;
}
```

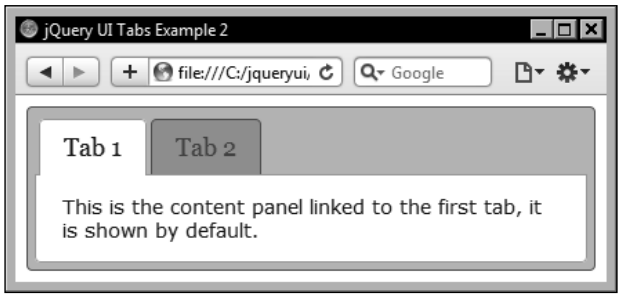
This is all we need. Save the file as `tabsTheme.css` in your `css` folder. If you compare the class names with the tables on the previous pages you'll see that we're overriding the theme-specific styles. Because we're overriding the theme file, we need to meet or exceed the specificity of the selectors in `theme.css`. This is why we target multiple selectors sometimes.

In this example we override some of the rules in `ui.tabs.css`. We need to use the ID selector of our container element along with the selector from `ui.theme.css` (`.ui-widget-content`) in order to beat the double class selector `.ui-tabs .ui-tabs-panel`.

Don't forget to link to the new stylesheet from the `<head>` of the underlying HTML file, and make sure the custom stylesheet we just created appears after the `ui.tabs.css` file:

```
<link rel="stylesheet" type="text/css" href="css/tabsTheme.css">
```

The rules that we are trying to override will be not overridden by our theme file if the stylesheets are not linked to in the correct order. Save the altered file as `tabs2.html` in the `jqueryui` folder and view it in a browser. It should look like the following screenshot:



Our new theme isn't dramatically different from the default smoothness. However, we can tweak its appearance to suit our own needs and preferences by adding just a few additional styles.

Configurable options

Each of the different components in the library has a series of different options that control which features of the widget are enabled by default. An object literal can be passed in to the `tabs` widget method to configure these options.

The available options to configure non-default behaviors when using the tabs widget are shown in the following table:

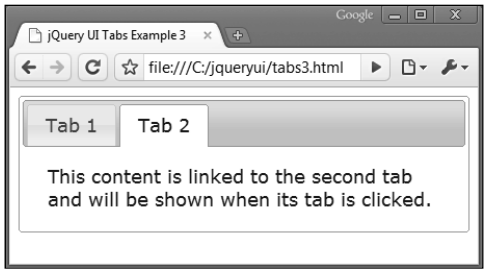
Option	Default value	Usage
`ajaxOptions`	`{}`	When using AJAX tabs and importing remote data into the tab panels, additional AJAX options are supplied via this property. We can use any of the options exposed by jQuery's `$.ajax` method such as `data`, `type`, `url`, and so on.
`cache`	`false`	Load remote tab content only once (lazy-load).
`collapsible`	`false`	Allows an active tab to be unselected if it is clicked.
`cookie`	`null`	Show active tab using cookie data on page load.
`disabled`	`[]`	Disable specified tabs on page load. Supply an array of index numbers to disable specific tabs.
`event`	`"click"`	The tab event that triggers the display of content.
`fx`	`null`	Specify an animation effect when changing tabs. Supply a literal object or an array of animation effects.
`idPrefix`	`"ui-tabs-"`	Used to generate a unique ID and fragment identifier when a remote tab's link element has no `title` attribute.
`panelTemplate`	`"<div></div>"`	A string specifying the elements used for the content section of a dynamically created tab widget.
`selected`	`0`	The tab selected by default when the widget is rendered (overrides the `cookie` property).
`spinner`	`"Loading&#B230"`	Specify the loading spinner for remote tabs.
`tabTemplate`	`<li><a href="#{href}"><span>#{label}</span></a></li>`	A string specifying the elements used when creating new tabs dynamically. Notice that both an `<a>` and a `<span>` tag are created when new tabs are added by the widget. The `#{href}` and `#{label}` parts of the string are used internally by the widget and are replaced with actual values by the widget.

Selecting a tab

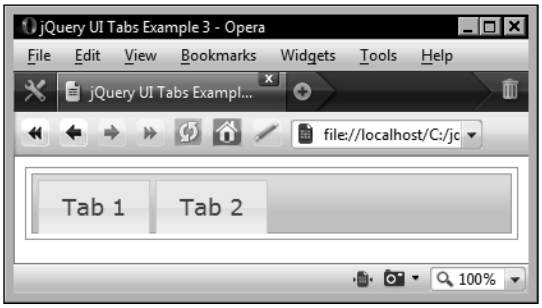
Let's look at how these configurable properties can be used. For example, let's configure the widget so that the second tab is displayed when the page loads. Remove the `<link>` for `tabsTheme.css` in the `<head>` and change the final `<script>` element so that it appears as follows:

```
<script type="text/javascript">
  $(function(){
    var tabOpts = {
      selected: 1
    };
    $("#myTabs").tabs(tabOpts);
  });
 </script>
```

Save this as `tabs3.html`. The different tabs and their associated content panels are represented by a numerical index starting at zero, much like a standard JavaScript array. Specifying a different tab to open by default is as easy as supplying its index number as the value for the `selected` property. When the page loads, the second tab should be selected.



We've switched to the default smoothness theme so that we can focus on how the properties work. Along with changing which tab is selected we can also specify that no tabs should be initially selected by supplying `null` as the value for this property. This will cause the widget to appear as follows on page load:



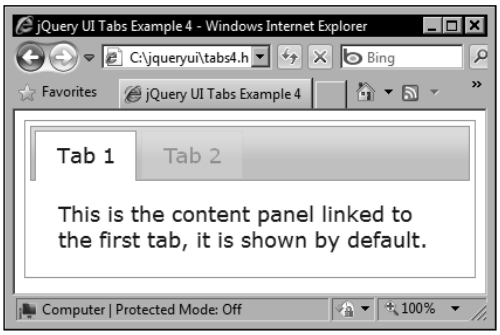
Disabling a tab

You may want a particular tab to be disabled until a certain condition is met. This is easily achieved by manipulating the `disabled` property of the tabs. Change the configuration object in `tabs3.html` to this:

```
var tabOpts = {
  disabled: [1]
};
```

Save this as `tabs4.html` in your `jqueryui` folder. In this example, we remove the `selected` property and add the index of the second tab to the `disabled` array. We could add the indices of other tabs to this array as well, separated by a comma, to disable multiple tabs by default.

When the page is loaded in a browser, the second tab has the class name `ui-widget-disabled` applied to it, and will pick up the disabled styles from `ui.theme.css`. It will not respond to mouse interactions in any way as shown in the following screenshot:



Transition effects

We can easily add attractive transition effects using the `fx` property. These are displayed when tabs are open and close. This property is configured using another object literal (or an array) inside our configuration object, which enables one or more effects. Let's enable fading effects using the following configuration object:

```
var tabOpts = {
  fx: {
    opacity: "toggle",
    duration: "slow"
  }
};
```

Save this file as `tabs5.html` in your `jqueryui` folder. The `fx` object that we created has two properties. The first property is the animation. To use fading, we specify `opacity` as this is what is adjusted. We would specify `height` as the property name instead to use opening animations. Toggling the `opacity` simply reverses its current setting. If it is currently visible, it is made invisible and vice-versa.

The second property, `duration`, specifies the speed at which the animation occurs. The values for this property are `slow`, `normal` (default value), or `fast`. We can also supply an integer representing the number of milliseconds the animation should run for.

When we run the file we can see that the tab content slowly fades out as a tab closes and fades in when a new tab opens. Both animations occur during a single tab interaction. To only show the animation once, when a tab closes for example, we would need to nest the `fx` object within an array. Change the configuration object in `tabs5.html` so that it appears as follows:

```
var tabOpts = {
        fx: [{
          opacity: "toggle",
          duration: "slow"
        },
        null]
};
```

The closing effect of the currently open content panel is contained within an object in the first item of the array, and the opening animation of the new tab is the second. By specifying `null` as the second item in the array we disable the opening animations when a new tab is selected.

We can also specify different animations and speeds for opening and closing animations by adding another object as the second array item instead of `null`. Save this as `tabs6.html` and view the results in a browser.

Collapsible tabs

By default when the currently active tab is clicked, nothing happens. But we can change this so that the currently open content panel closes when its tab heading is selected. Change the configuration object in `tabs6.html` so that it appears as follows:

```
var tabOpts = {
  collapsible: true
};
```

Save this version as `tabs7.html`. This option allows all of the content panels to be closed, much like when we supplied `null` to the `selected` property earlier on. Clicking a deactivated tab will select the tab and show its associated content panel. Clicking the same tab again will close it, shrinking the widget down so that only the header and tabs are displayed).

Tab events

The tab widget defines a series of useful options that allow you to add callback functions to perform different actions when certain events exposed by the widget are detected. The following table lists the configuration options that are able to accept executable functions on an event:

Option	Usage
`add`	Execute a function when a new tab is added.
`disable`	Execute a function when a tab is disabled.
`enable`	Execute a function when a tab is enabled.
`load`	Execute a function when a tab's remote data has loaded.
`remove`	Execute a function when a tab is removed.
`select`	Execute a function when a tab is selected.
`show`	Execute a function when the content section of a tab is shown.

Each component of the library has callback options (such as those in the previous table), which are tuned to look for key moments in any visitor interaction. Any function we use with these callbacks are usually executed *before* the change happens. Therefore, you can return `false` from your callback and prevent the action from occurring.

In our next example, we will look at how easy it is to react to a particular tab being selected using the standard non-bind technique. Change the final `<script>` element in `tabs7.html` so that it appears as follows:

```
<script type="text/javascript">
  $(function(){

    function handleSelect(event, tab) {
      $("<p>").text("The tab at index " + tab.index +
        " was selected").addClass("status-message ui-corner-all").
        appendTo($(".ui-tabs-nav", "#myTabs")).fadeOut(5000);
     }
```

```
    var tabOpts = {
      select:handleSelect
    };

    $("#myTabs").tabs(tabOpts);
  });
</script>
```

Save this file as `tabs8.html`. We also need a little CSS to complete this example, in the `<head>` of the page we just created add the following `<link>` element:

```
<link rel="stylesheet" type="text/css" href="css/tabSelect.css">
```

Then in a new page in your text editor add the following code:

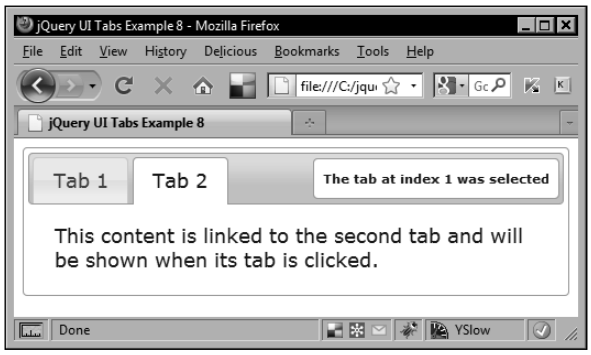
```
.status-message {
  position:absolute; right:3px; top:4px; margin:0;
  padding:11px 8px 10px; font-size:11px;
  background-color:#ffffff; border:1px solid #aaaaaa;
}
```

Save this file as `tabSelect.css` in the `css` folder.

We made use of the `select` callback in this example, although the principle is the same for any of the other custom events fired by tabs. The name of our callback function is provided as the value of the `select` property in our configuration object.

Two arguments will be passed automatically to the function we define by the widget when it is executed. These are the original event object and a custom object containing useful properties from the tab which is in the function's execution context.

To find out which of the tabs was clicked, we can look at the `index` property of the second object (remember these are zero-based indices). This is added, along with a little explanatory text, to a paragraph element that we create on the fly and append to the widget header.



In this example, the callback function was defined outside the configuration object, and was instead referenced by the object. We can also define these callback functions inside our configuration object to make our code more efficient. For example, our function and configuration object from the previous example could have been defined like this:

```
var tabOpts = {
  select: function(event, tab) {
    $("<p>").text("The tab at index " + tab.index +
      " was selected").addClass("status-message ui-corner-all").
      appendTo($(".ui-tabs-nav", "#myTabs")).fadeOut(5000);
  }
}
```

Check `tabs8inline.html` in the code download for further clarification on this way of using event callbacks. Whenever a tab is selected, you should see the paragraph before it fades away. Note that the event is fired before the change occurs.

Binding to events

Using the event callbacks exposed by each component is the standard way of handling interactions. However, in addition to the callbacks listed in the previous table we can also hook into another set of events fired by each component at different times.

We can use the standard jQuery `bind()` method to bind an event handler to a custom event fired by the tabs widget in the same way that we could bind to a standard DOM event, such as a `click`.

The following table lists the tab widget's custom binding events and their triggers:

Event	Trigger
`tabsselect`	A tab is selected.
`tabsload`	A remote tab has loaded.
`tabsshow`	A tab is shown.
`tabsadd`	A tab has been added to the interface.
`tabsremove`	A tab has been removed from the interface.
`tabsdisable`	A tab has been disabled.
`tabsenable`	A tab has been enabled.

The first three events are fired in succession in the order event in which they appear in the table. If no tabs are remote then `tabsselect` and `tabsshow` are fired in that order. These events are sometimes fired before and sometimes after the action has occurred, depending on which event is used.

Let's see this type of event usage in action, change the final `<script>` element in `tabs8.html` to the following:

```
<script type="text/javascript">
  $(function() {
    $("#myTabs").tabs();
    $("#myTabs").bind("tabsselect", function(e, tab) {
      alert("The tab at index " + tab.index + " was selected");
    });
  });
</script>
```

Save this change as `tabs9.html`. Binding to the `tabsselect` in this way produces the same result as the previous example using the `select` callback function. Like last time, the alert should appear before the new tab is activated.

All the events exposed by all the widgets can be used with the `bind()` method, by simply prefixing the name of the widget to the name of the event.

Using tab methods

The tabs widget contains many different methods, which means it has a rich set of behaviors. It also supports the implementation of advanced functionality that allows us to work with it programmatically. Let's take a look at the methods which are listed in the following table:

Method	Usage
`abort`	Stops any animations or AJAX requests that are currently in progress.
`add`	Add a new tab programmatically, specifying the URL of the tab's content, a label, and optionally its index number as arguments.
`destroy`	Completely remove the tabs widget.
`disable`	Disable a tab based on index number.
`enable`	Enable a disabled tab based on index number.
`length`	Return the number of tabs in the widget.
`load`	Reload an AJAX tab's content, specifying the index number of the tab.
`option`	Get or set any property after the widget has been initialized.
`remove`	Remove a tab programmatically, specifying the index of the tab to remove.
`rotate`	Automatically changes the active tab after a specified number of milliseconds have passed, either once or repeatedly.
`select`	Select a tab programmatically, which has the same effect as when a visitor clicks a tab, based on index number.
`url`	Change the URL of content given to an AJAX tab. The method expects the index number of the tab and the new URL. See also `load` (above).

Enabling and disabling tabs

We can make use of the `enable` or `disable` methods to programmatically enable or disable specific tabs. This will effectively switch on any tabs that were initially disabled or disable those that are currently active. Let's use the `enable` method to switch on a tab, which we disabled by default in an earlier example. Add the following new `<button>` directly after the markup for the tabs widget in `tabs4.html`:

```
<button id="enable">Enable!<
   /button><button id="disable">Disable!</button>
```

Next change the final <script> element so that it appears as follows:

```
<script type="text/javascript">
  $(function(){
    var tabOpts = {
      disabled:[1]
    };
    $("#myTabs").tabs(tabOpts);
    $("#enable").click(function() {
      $("#myTabs").tabs("enable", 1);
    });
    $("#disable").click(function() {
      $("#myTabs").tabs("disable", 1);
    });
  });
</script>
```

Save the changed file as `tabs10.html`. On the page we've added two new `<button>` elements—one will be used to enable the disabled tab and the other used to disable it again.

In the JavaScript, we use the `click` event of the **Enable!** button to call the `tabs` constructor. This passes the string `"enable"`, which specifies the `enable` method and the index number of the tab we want to enable. The `disable` method is used in the same way. Note that a tab cannot be disabled while it is active.

All methods exposed by each component are used in this same easy way which you'll see more of as we progress through the book.

I mentioned in *Chapter 1* that each widget has a set of common methods consisting of `enable`, `disable`, and `destroy`. These methods are used in the same way across each of the different components, so we won't be looking at these methods again.

Adding and removing tabs

Along with enabling and disabling tabs programmatically, we can also remove them or add completely new tabs dynamically. In `tabs10.html` add the following new code directly after the underlying HTML of the widget:

```
<label>Enter a tab to remove:</label>
  <input id="indexNum"><button id="remove">Remove!</button><br>
<button id="add">Add a new tab!</button>
<div id="newTab" class="ui-helper-hidden"> This content was added
  after the widget was initialized!</div>
```

Then change the final `<script>` element to this:

```
<script type="text/javascript">
  $(function(){
    $("#myTabs").tabs();
    $("#remove").click(function() {
      var indexNumber = $("#indexNum").val();
      $("#myTabs").tabs("remove", indexNumber);
    });
    $("#add").click(function() {
      var newLabel = "A New Tab!"
      $("#myTabs").tabs("add", "#newTab", newLabel);
    });
  });
```

Save this as `tabs11.html`. On the page we've changed the `<button>` from the last example and have added a new `<label>`, an `<input>`, and another `<button>`. These new elements are used to add a new tab.

We have also added some new content on the page, which will be used as the basis for each new tab that is added. We make use of the `ui-helper-hidden` framework class to hide this content, so that it isn't available when the page loads,. Even though this class name will remain on the element once it has been added to the tab widget, it will still be visible when its tab is clicked. This is because the class name will be overridden by classes within `ui.tabs.css`.

In the `<script>`, the first of our new functions handles removing a tab using the `remove` method. This method requires one additional argument—the index number of the tab to be removed. In this example, we get the value entered into the text box and pass it to the method as the argument. If no index is passed to the method, the first tab will be removed.

The `add` method that adds a new tab to the widget, can be made to work in several different ways. In this example, we've specified that content already existing on the page (the `<div>` with an `id` of `newTab`) should be added to the tabs widget. In addition to passing the string `"add"` and specifying a reference to the element we wish to add to the tabs, we also specify a label for the new tab.

Optionally, we can also specify the index number where the new tab should be inserted. If the index is not supplied, the new tab will be added as the last tab. We can continue adding new tabs and each one will reuse the `<div>` for its content because our content `<div>` will retain its `id` attribute after it has been added to the widget. After adding and perhaps removing tabs, the page should appear something like this:

Simulating clicks

There may be times when you want to programmatically select a particular tab and show its content. This could happen as the result of some other interaction by the visitor. We can use the `select` method to do this, which is completely analogous with the action of clicking a tab. Alter the final `<script>` block in `tabs11.html` so that it appears as follows:

```
<script type="text/javascript">
  $(function(){
    $("#myTabs").tabs();
    $("#remove").click(function() {
      var indexNumber = $("#indexNum").val() - 1;
      $("#myTabs").tabs("remove", indexNumber);
    });
    $("#add").click(function() {
      var newLabel = "A New Tab!"
```

```
        $("#myTabs").tabs("add", "#newTab", newLabel);
        var newIndex = $("#myTabs").tabs("length") - 1;
        $("#myTabs").tabs("select", newIndex);
      });
    });
</script>
```

Save this as `tabs12.html` in your `jqueryui` folder. Now when a new tab is added, it is automatically selected. The `select` method requires just one additional parameter, which is the index number of the tab to select.

As any tab we add will be the last tab in the interface (in this example) and as the tab indices are zero based, all we have to do is use the `length` method to return the number of tabs and then subtract 1 from this figure to get the index. The result is passed to the `select` method.

Creating a tab carousel

One method that creates quite an exciting result is the `rotate` method. The `rotate` method will make all of the tabs (and their associated content panels) display one after the other automatically.

It's a great visual effect and is useful for ensuring that all, or a lot, of the individual tab's content panels get seen by the visitor. For an example of this kind of effect in action, see the homepage of `http://www.cnet.com`. There is a tabs widget (not a jQuery UI one) that shows blogs, podcasts, and videos.

Like the other methods we've seen, the `rotate` method is easy to use. Change the final `<script>` element in `tabs9.html` to this:

```
<script type="text/javascript">
  $(function(){
    $("#myTabs").tabs().tabs("rotate", 1000, true);
  });
</script>
```

Save this file `as tabs13.html`. We've reverted back to a simplified page with no additional elements other than the underlying structure of the widget. Although we can't call the `rotate` method directly using the initial `tabs` method, we can chain it to the end like we would with methods from the standard jQuery library.

Chaining UI Methods

Chaining widget methods is possible because like the methods found in the underlying jQuery library, they always return the jQuery ($) object.

The `rotate` method is used with two additional parameters. The first parameter is an integer, that specifies the number of milliseconds each tab should be displayed before the next tab is shown. The second parameter is a Boolean that indicates whether the cycle through the tabs should occur once or continuously.

The tab widget also contains a `destroy` method. This is a method common to all the widgets found in jQuery UI. Let's see how it works. In `tabs13.html`, after the widget add a new `<button>` as follows:

```
<button id="destroy">Destroy the tabs!</button>
```

Next change the final `<script>` element to this:

```
<script type="text/javascript">
  $(function(){
    $("#myTabs").tabs();
    $("#destroy").click(function() {
      $("#myTabs").tabs("destroy");
    });
  });
</script>
```

Save this file as `tabs14.html`. The `destroy` method that we invoke with a click on the button, completely removes the tab widget, returning the underlying HTML to its original state. After the button has been clicked, you should see a standard HTML list element and the text from each tab, just like in the following screenshot:

Once the tabs have been reduced to this state it would be common practice to remove them using jQuery's `remove()` method. As I mentioned with the `enable` and `disable` methods earlier, the `destroy` method is used in exactly the same way for all widgets and therefore will not be discussed again.

Getting and setting options

Like the `destroy` method the `option` method is exposed by all the different components found in the library. This method is used to work with the configurable options and functions in both getter and setter modes. Let's look at a basic example, add the following `<button>` after the tabs widget in `tabs9.html`:

```
<button id="show">Show Selected!</button>
```

Then change the final `<script>` element so that it is as follows:

```
<script type="text/javascript">
  $(function(){
   $("#myTabs").tabs();
     $("#show").click(function() {
       $("<p>").text("The tab at index " + $("#myTabs").
         tabs("option", "selected") + " is active").addClass(
         "status-message ui-corner-all").appendTo($(".ui-tabs-nav",
         "#myTabs")).fadeOut(5000);
    });
  });
</script>
```

Save this file as `tabs15.html`. The `<button>` on the page has been changed so that it shows the currently active tab. All we do is add the index of the selected tab to a status bar message as we did in the earlier example. We get the `selected` option by passing the string `selected` as the second argument. Any option can be accessed in this way.

To trigger setter mode instead, we can supply a third argument containing the new value of the option that we'd like to set. Therefore, to change the value of the `selected` option, we could use the following HTML to specify the tab to select:

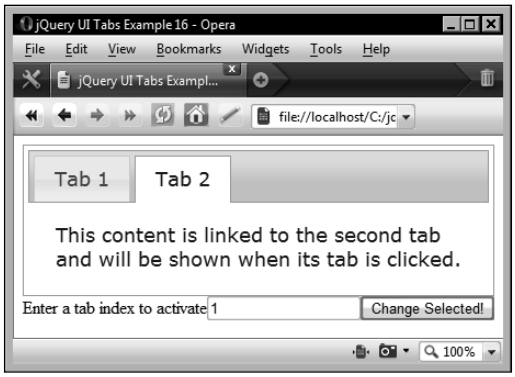
```
<label>Enter a tab index to activate</label><input id="newIndex"
   type="text"><button id="set">Change Selected!</button>
```

And the following click-handler:

```
<script type="text/javascript">
  $(function(){
$("#set").click(function() {
  $("#myTabs").tabs("option", "selected", parseInt($("#newIndex").
    val()));
});
```

Save this as `tabs16.html`. The new page contains a `<label>` and an `<input>`, as well as a `<button>` that is used to harvest the index number that the `selected` option should be set to. When the button is clicked, our code will retrieve the value of the `<input>` and use it to change the `selected` index. By supplying the new value we put the method in setter mode.

When we run this page in our browser, we should see that we can switch to the second tab by entering its index number and clicking the **Changed Selected** button.



AJAX tabs

We've looked at adding new tabs from already existing content on the page. In addition to this we can also create AJAX tabs that load content from remote files or URLs. Let's extend our previous example of adding tabs so that the new tab content is loaded from an external file. In `tabs16.html` remove the `<label>` and the `<input>` from the page and change the `<button>` so that it appears as follows:

```
<button id="add">Add a new tab!</button>
```

Then change the click-handler so that it appears as follows:

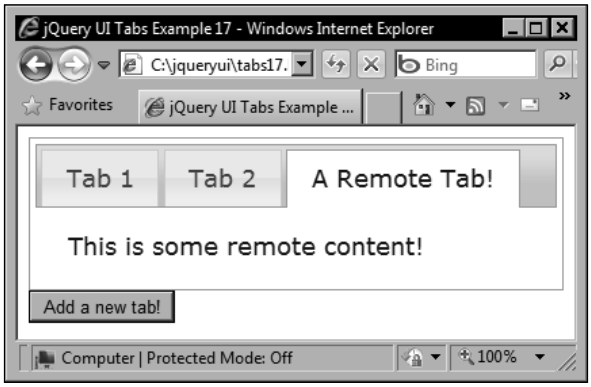
```
$("#add").click(function() {
  $("#myTabs").tabs("add", "tabContent.html", "A Remote Tab!");
});
```

Save this as `tabs17.html`. This time, instead of specifying an element selector as the second argument of the `add` method, we supply a relative file path. Instead of generating the new tab from inline content, the tab becomes an AJAX tab and loads the contents of the remote file.

The file used as the remote content in this example is basic and consists of just the following code:

```
<div>This is some remote content!</div>
```

Save this as `tabContent.html` in the `jqueryui` folder. After the `<button>` has been clicked, the page should appear like this:



Instead of using JavaScript to add the new tab, we can use plain HTML to specify an AJAX tab as well. In this example, we want the tab that will display the remote content to be available all the time, not just after clicking the button. Add the following new `<a>` element to the underlying HTML for the widget in `tabs17.html`:

```
<li><a href="tabContent.html">AJAX Tab</a></li>
```

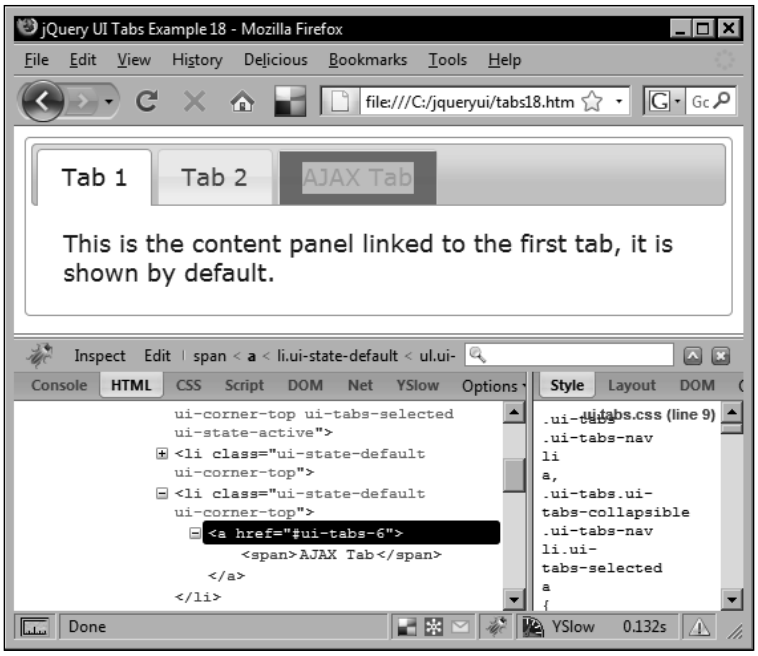
The final `<script>` element can be used to just call the `tabs` method:

```
$("#myTabs").tabs();
```

Save this as `tabs18.html`. All we're doing is specifying the path to the remote file (the same one we created in the previous example) using the `href` attribute of an `<a>` element in the underlying markup from which the tabs are created.

Unlike static tabs, we don't need a corresponding `<div>` element with an `id` that matches the `href` of the link. The additional elements required for the tab content will be generated automatically by the widget.

If you use a DOM explorer, you can see that the file path we added to link to the remote tab has been removed. Instead, a new fragment identifier has been generated and set as the `href`. The new fragment is also added as the `id` of the new tab (minus the `#` symbol of course).



There is no inherent cross-domain support built into the AJAX functionality of tabs widget. Therefore, unless additional PHP or some other server-scripting language is employed as a proxy, or you wish to make use of JSON structured data and jQuery's JSONP functionality, files and URLs should be under the same domain as the page running the widget.

Along with loading data from external files, it can also be loaded from URLs. This is great when retrieving content from a database using query strings or a web service. Methods related to AJAX tabs include the `load` and `url` methods. The `load` method is used to load and reload the contents of an AJAX tab, which could come in handy for refreshing content that changes very frequently.

The `url` method is used to change the URL that the AJAX tab retrieves its content from. Let's look at a brief example of these two methods in action. There are also a number of properties related to AJAX functionality. Add the following new `<select>` element in `tabs18.html`:

```
<select id="fileChooser">
  <option>tabContent.html</option>
  <option>tabContent2.html</option>
</select>
```

Then change the final `<script>` element to this:

```
<script type="text/javascript">
  $(function(){
    $("#myTabs").tabs();
    $("#fileChooser").change(function() {
      this.selectedIndex == 0 ? loadFile1() : loadFile2();
      function loadFile1() {
        $("#myTabs").tabs("url", 2, "tabContent.html").tabs(
          "load", 2);
      }
      function loadFile2() {
        $("#myTabs").tabs("url", 2, "tabContent2.html").tabs(
          "load", 2);
      }
    });
  });
</script>
```

Save the new file as `tabs19.html`. We've added a simple `<select>` element to the page that lets you choose the content to display in the AJAX tab. In the JavaScript, we set a `change` handler for the `<select>` and specified an anonymous function to be executed each time the event is detected.

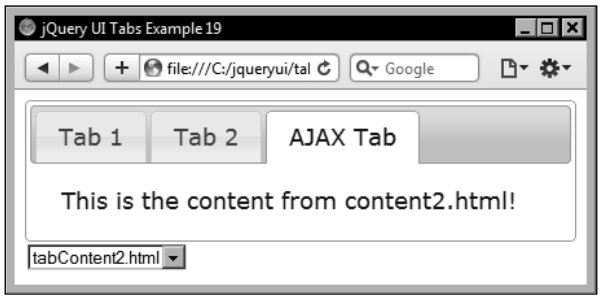
This function checks the `selectedIndex` of the `<select>` element and calls either the `loadFile1` or `loadFile2` function. The `<select>` element is in the execution scope of the function, so we can refer to it using the `this` keyword.

These functions are where things get interesting. We first call the `url` method, specifying two additional arguments, which are the index of the tab whose URL we want to change followed by the new URL. We then call the `load` method that is chained to the `url` method, specifying the index of the tab whose content we want to load.

We'll need a second local content file, change the text on the page of `tabContent1.html` and resave it as `tabContent2.html`.

Run the new file in a browser and select a tab. Then use the dropdown `<select>` to choose the second file and watch as the content of the tab is changed. You'll also see that the tab content will be reloaded even if the AJAX tab isn't active when you use the `<select>` element.

The slight flicker in the tab heading is the string value of the `spinner` option that by default is set to `Loading....` Although, we don't get a chance to see it in full as the tab content is changed quickly when running it locally. Here's how the page should look after selecting the remote page in the dropdown select and the third tab:



Displaying data obtained via JSONP

Let's pull in some external content for our final tabs example. If we use the tabs widget, in conjunction with the standard jQuery library `getJSON` method, we can bypass the cross-domain exclusion policy and pull in a feed from another domain to display in a tab. In a new file in your text editor, create the following new page:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<html lang="en">
  <head>
    <link rel="stylesheet" type="text/css"
      href="development-bundle/themes/smoothness/ui.all.css">
    <link rel="stylesheet" type="text/css"
      href="css/flickrTabTheme.css">
    <meta http-equiv="Content-Type"
      content="text/html; charset=utf-8">
    <title>jQuery UI AJAX Tabs Example</title>
  </head>
  <body>
    <div id="myTabs">
      <ul>
        <li><a href="#a"><span>Nebula Information</span></a></li>
        <li><a href="#flickr"><span>Images</span></a></li>
      </ul>
      <div id="a">
```

```
        <p>A nebulae is an interstellar cloud of dust, hydrogen gas,
and plasma. It is the first stage of a star's cycle. In these regions
the formations of gas, dust, and other materials clump together to
form larger masses, which attract further matter, and eventually will
become big enough to form stars. The remaining materials are then
believed to form planets and other planetary system objects. Many
nebulae form from the gravitational collapse of diffused gas in the
interstellar medium or ISM. As the material collapses under its own
weight, massive stars may form in the center, and their ultraviolet
radiation ionizes the surrounding gas, making it visible at optical
wavelengths.</p>
      </div>
      <div id="flickr"></div>
    </div>
    <script type="text/javascript"
      src="development-bundle/jquery-1.3.2.js"></script>
    <script type="text/javascript"
      src="development-bundle/ui/ui.core.js"></script>
    <script type="text/javascript"
      src="development-bundle/ui/ui.tabs.js"></script>
  </body>
</html>
```

The HTML seen here is nothing new. It's basically the same as the previous examples so I won't describe it in any detail. The only point worthy noting is that unlike the previous AJAX tab examples, we have specified an empty `<div>` element that will be used for the AJAX tab's content. Now, just before the `</body>` tag, add the following script block:

```
<script type="text/javascript">
  $(function(){
    var tabOpts = {
      select: function(event, ui) {
        ui.tab.toString().indexOf("flickr") != -1 ? getData() : null ;
        function getData() {
          $("#flickr").empty();
    $.getJSON("http://api.flickr.com/services/feeds/photos_public.gne?
     tags=nebula&format=json&jsoncallback=?", function(data) {
            $.each(data.items, function(i,item){
             $("<img/>").attr("src",
             item.media.m).appendTo("#flickr").height(100).width(100);
              return (i == 5) ? false : null;
            });
          });
        }
      }
```

```
        }
        $("#myTabs").tabs(tabOpts);
      });
    </script>
```

Save the file as `flickrTab.html` in your `jqueryui` folder. Every time a tab is selected, our `select` callback will check to see if it was the tab with an `id` of `flickr` that was clicked. If it is, then the `getData()` function is invoked that uses the standard jQuery `getJSON` method to retrieve an image feed from `http://www.flickr.com`.

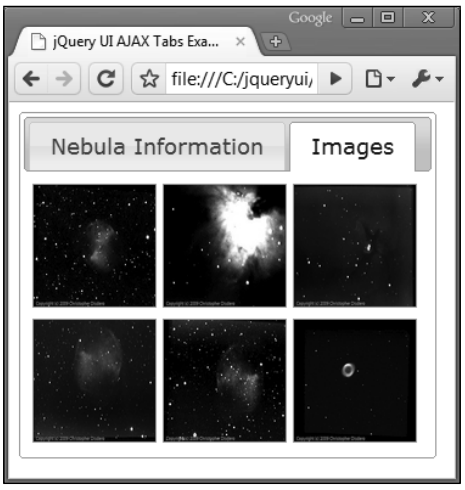
Once the data is returned, the anonymous callback function iterates over each object within the feed and creates a new image. We also remove any preexisting images from the content panel to prevent a buildup of images following multiple tab selections.

Each new image has its `src` attribute set using the information from the current feed object and is then added to the empty Flickr tab. Once iteration over six of the objects in the feed has occurred, we exit jQuery's `each` method. It's that simple.

We also require a bit of CSS to make the example look right. In a new file in your text editor add the following selectors and rules:

```
#myTabs { width:335px; }
#myTabs .ui-tabs-panel { padding:10px 0 0 7px; }
#myTabs p { margin:0 0 10px; font-size:75%; }
#myTabs img { border:1px solid #aaaaaa; margin:0 5px 5px 0; }
```

Save this as `flickrTabTheme.css` in your `css` folder. When you view the page and select the **Images** tab, after a short delay you should see six new images, as seen in the following screenshot:



Summary

The tabs widget is an excellent way of saving space on your page by organizing related (or even completely unrelated) sections of content that can be shown or hidden, with simple click-input from your visitors. It also lends an air of interactivity to your site that can help improve the overall functionality and appeal of the page on which it is used.

Let's review what was covered in this chapter. We first looked at how, with just a little underlying HTML and a single line of jQuery-flavored JavaScript code, we can implement the default tabs widget.

We then saw how easy it is to add our own basic styling for the tabs widget so that its appearance, but not its behavior, is altered. We already know that in addition to this we can use a predesigned theme or create a completely new theme using ThemeRoller.

We then moved on to look at the set of configurable options exposed by the tabs API. With these, we can enable or disable different options that the widget supports, such as whether tabs are selected by clicks or another event, whether certain tabs are disabled when the widget is rendered, and so on.

We took some time to look at how we can use a range of predefined callback options that allow us to execute arbitrary code when different events are detected. We also saw that the jQuery `bind()` method can listen for the same events if it becomes necessary.

Following the configurable options, we covered the range of methods that we can use to programmatically make the tabs perform different actions, such as simulating a click on a tab, enabling or disabling a tab, and adding or removing tabs.

We briefly looked at some of the more advanced functionality supported by the tabs widget such as AJAX tabs and the tab carousel. Both these techniques are easy to use and can add value to any implementation.

Where to buy this book

You can buy jQuery UI 1.7 from the Packt Publishing website:
`http://www.packtpub.com/user-interface-library-for-jquery-ui-1-7/book`

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

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

www.PacktPub.com