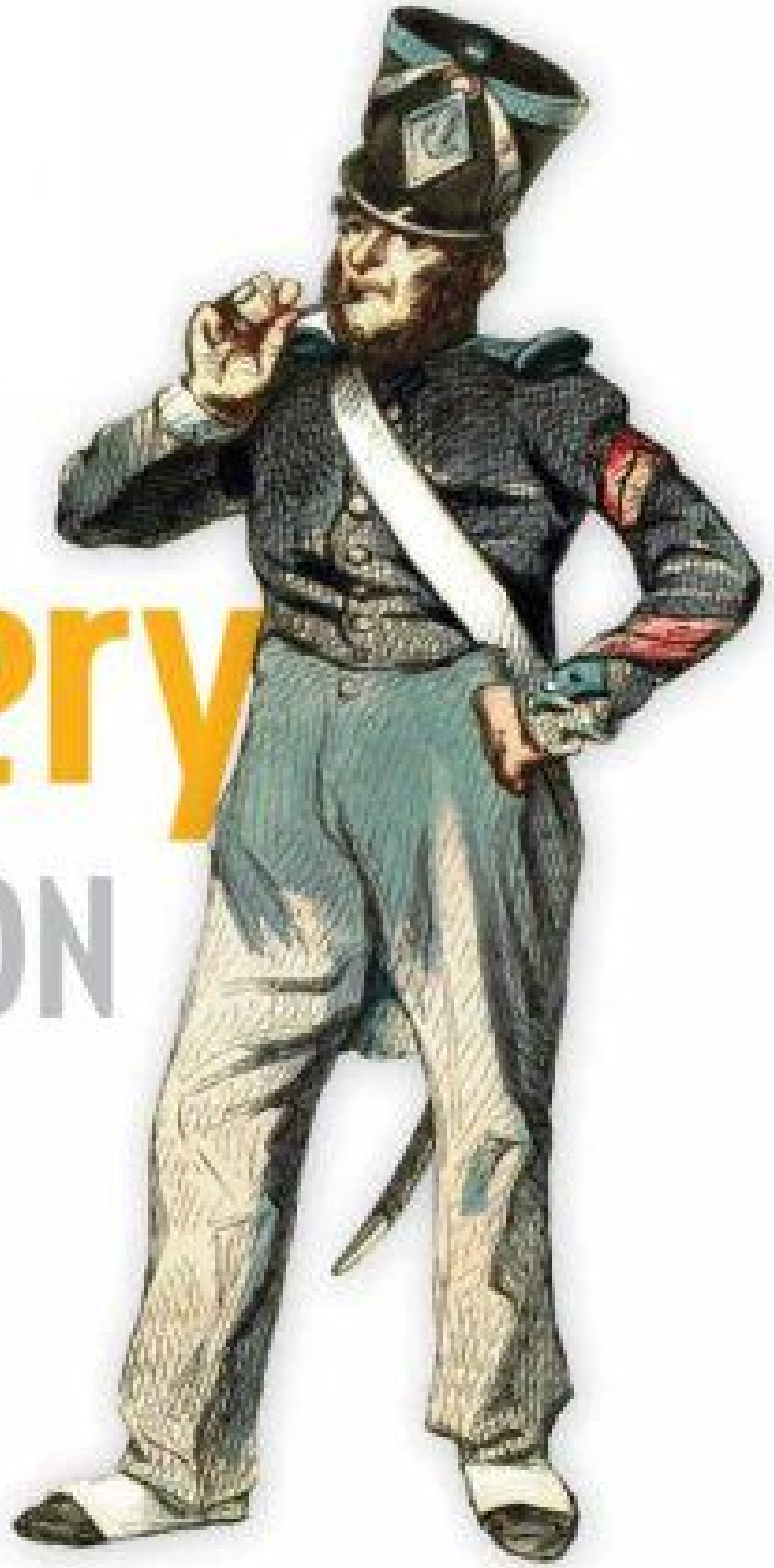
Bear Bibeault
Yehuda Katz

Covers jQuery 1.4 and jQuery UI 1.8



jQuery IN ACTION

SECOND EDITION

MANNING

Table of Contents

11 jQuery UI widgets: Beyond HTML controls

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 resemblance? 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.

11.1 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.

11.1.1 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.

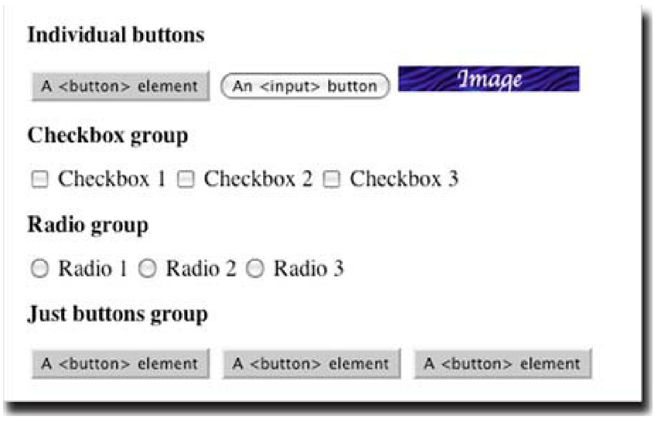


Figure 11.1 Various button elements without any styling—rather boring, wouldn't you say?

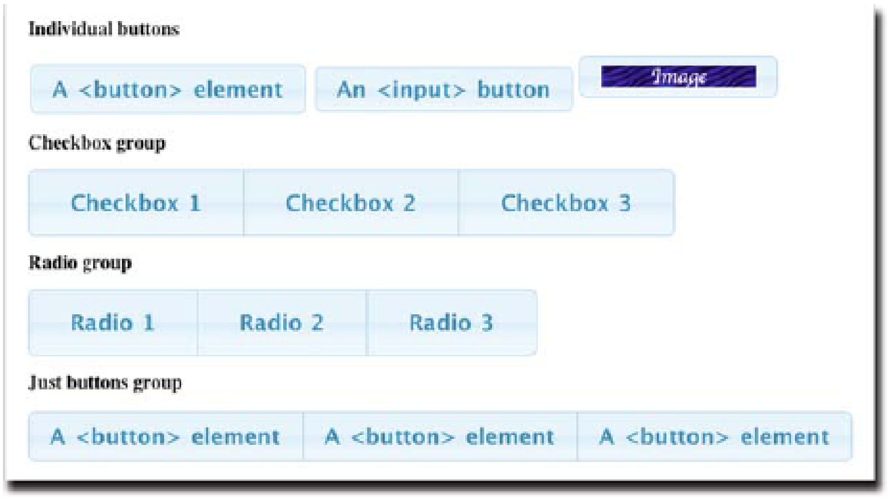


Figure 11.2 After a style makeover, our buttons are dressed in their best and ready to hit the town!

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.

11.1.2 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(options)`
`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 `<span>` and `<div>`.

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:

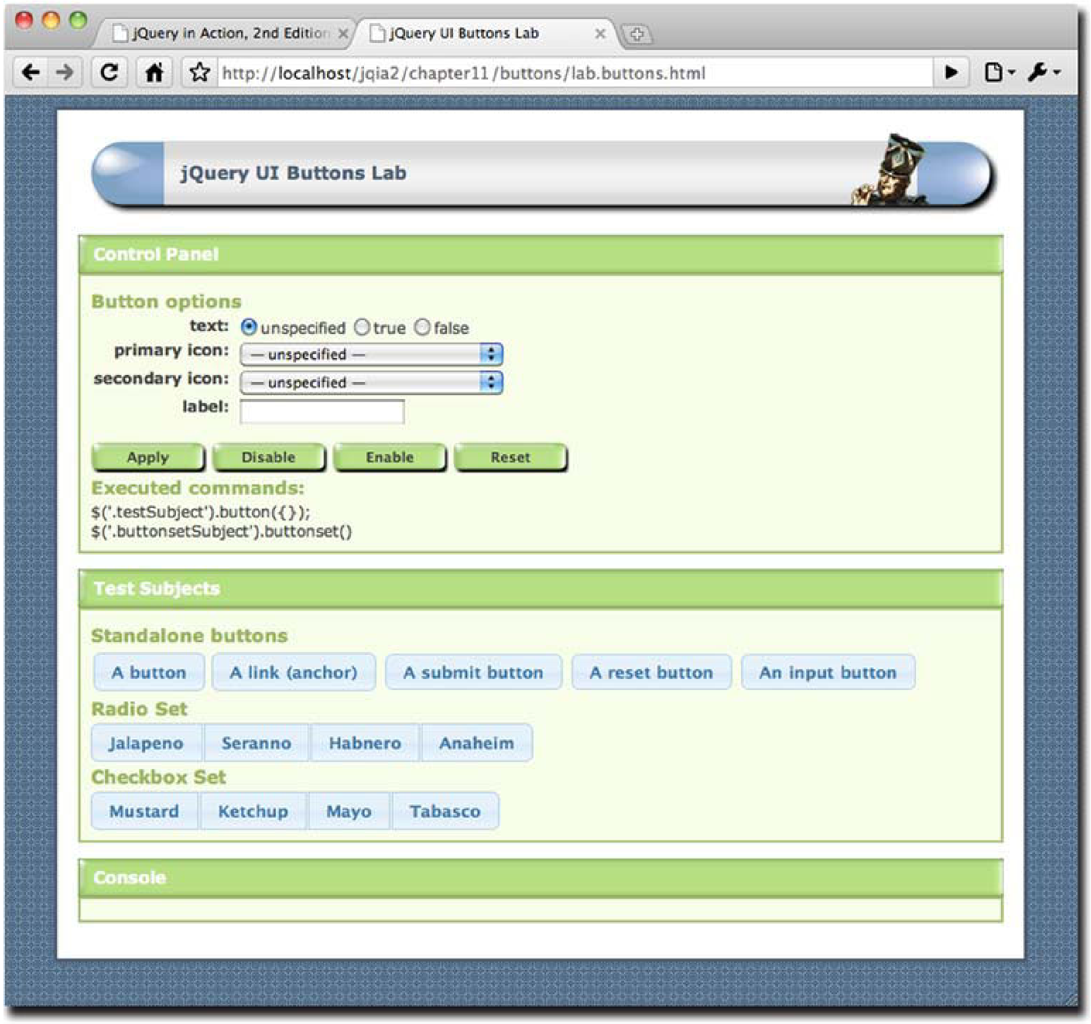
```
$(':button').button({ text: true });
```

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.



```
$('.testSubject').button({});
$('.buttonsetSubject').buttonset()
```

Figure 11.3 The jQuery UI Buttons Lab page lets us see the before and after states of buttons, as well as fiddle with the options.

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.

11.1.3 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.

11.1.4 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.

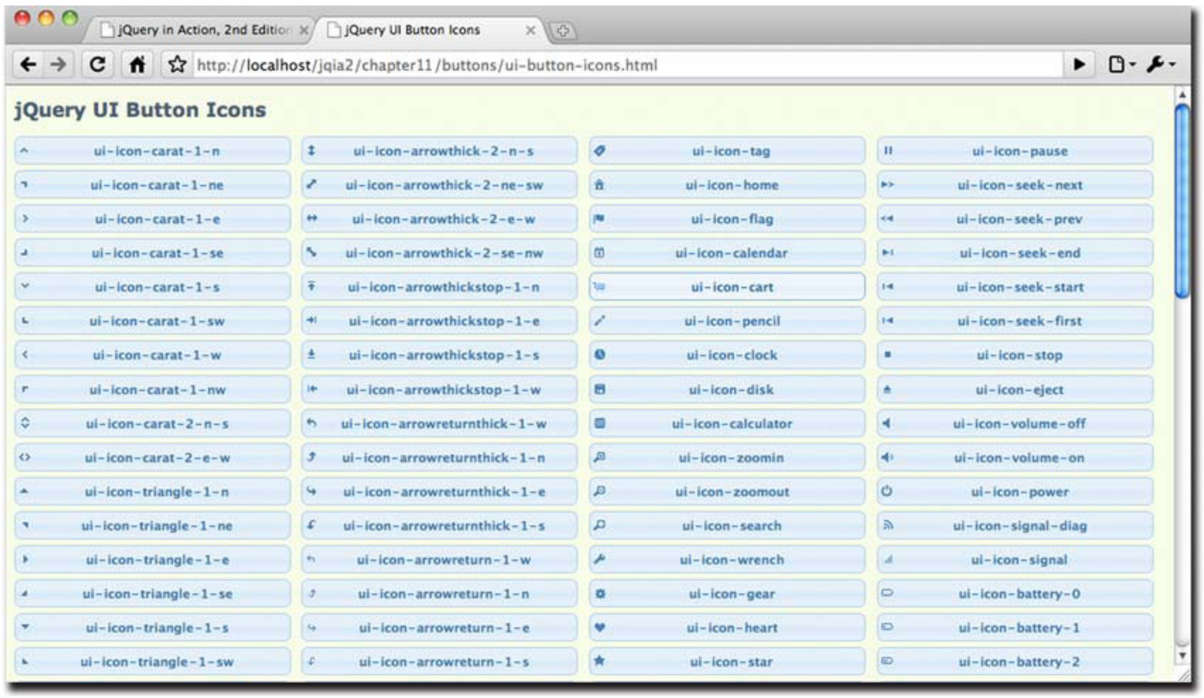


Figure 11.4 The jQuery UI Button Icons page lets us see all the available button icons along with their names.

11.1.5 *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 `<span>` 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.

11.2 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.

11.2.1 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 `<div>` and `<a>`. The target `<div>` 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.

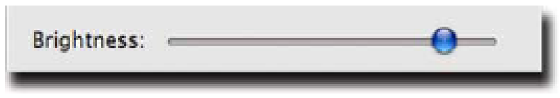


Figure 11.5 Sliders can be used to represent a range of values; in this example, from minimum to full brightness.

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 (`<div>` elements recommended) into a slider control.

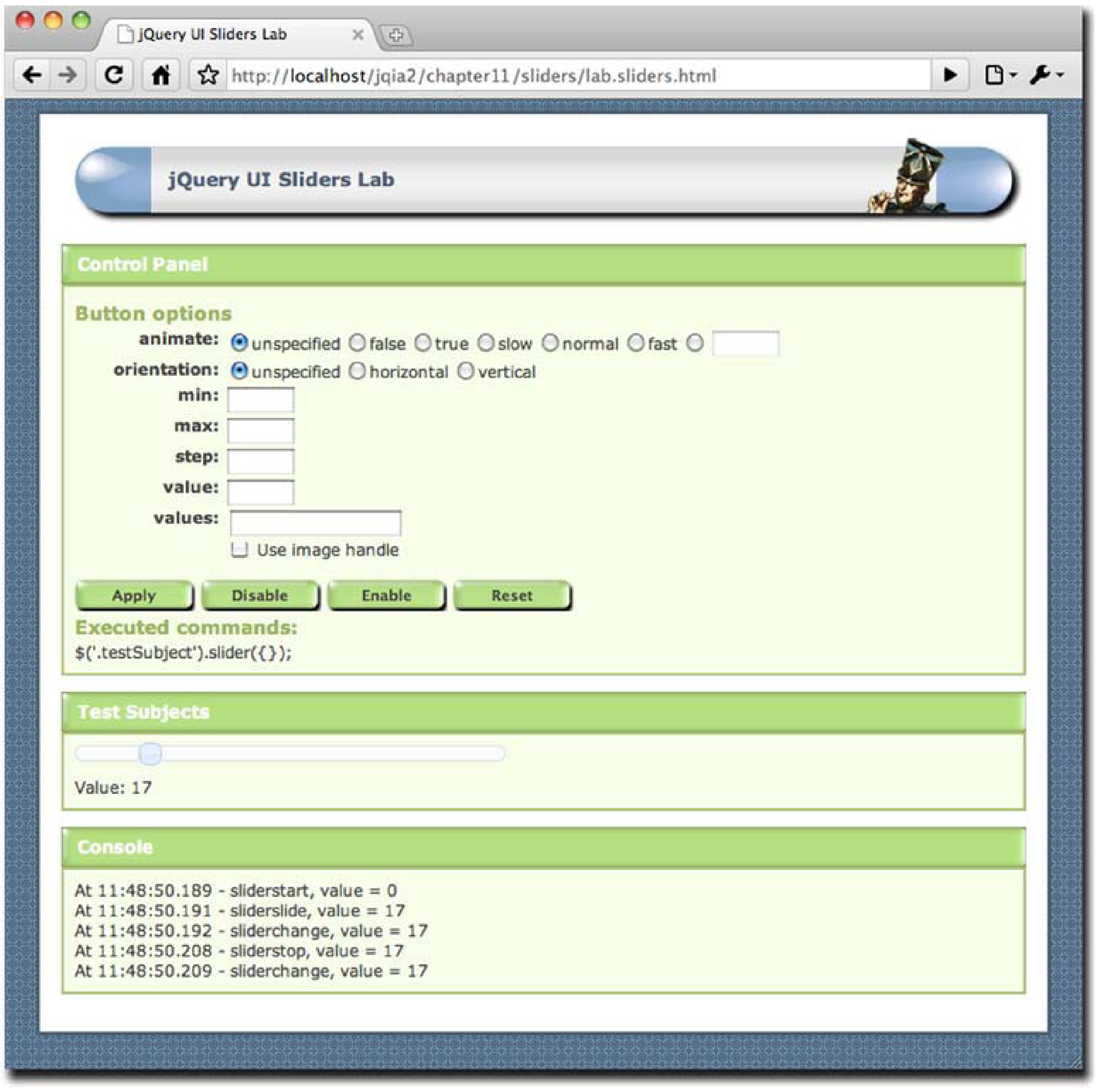
Parameters

`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 new values 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.



```
$('.testSubject').slider({});
```

Value: 17

```
At 11:48:50.189 - sliderstart, value = 0
At 11:48:50.191 - sliderslide, value = 17
At 11:48:50.192 - sliderchange, value = 17
At 11:48:50.208 - sliderstop, value = 17
At 11:48:50.209 - sliderchange, value = 17
```

Figure 11.6 The jQuery UI Sliders Lab shows the various ways that jQuery UI sliders can be set up and manipulated.

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.	✓

Table 11.2 Options for the jQuery UI sliders *(continued)*

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.	✓

11.2.2 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`	`slide`	Triggered for mousemove events whenever the handle is being dragged through the trough. Returning `false` cancels the slide.
`slidechange`	`change`	Triggered whenever a handle's value changes, either through user action or programmatically.
`slidestart`	`start`	Triggered when a slide starts.
`slidestop`	`stop`	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;
}
```

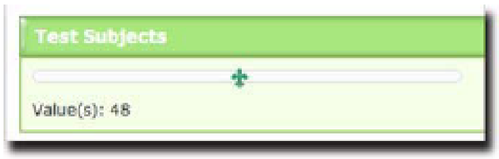


Figure 11.7 With a PNG image and a little CSS magic, we can make the slider handle look like whatever we want.

The class name `testSubject` is the class that's used within the Lab to identify the element to be transformed into the slider.

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.

11.3 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.

11.3.1 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 (`<div>` 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.

11.3.2 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 on the page that shows the exact completion percentage of the control, or for any other reason 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.

11.3.3 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 (`<div>` 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(){
  var bar$ = $(this);
  bar$.data(
    'autoProgressbar-interval',
    window.setInterval(function(){
      $.ajax({
        url: settings.pulseUrl,
        data: settings.pulseData,
        global: false,
        dataType: 'json',
        success: function(value){
          if (value != null) bar$.autoProgressbar('value',value);
          if (value == 100) bar$.autoProgressbar('stop');
        }
      });
    },settings.interval));
});
```

1 Iterates over wrapped set

2 Stores interval handle on widget

3 Starts interval timer

4 Fires off Ajax request

5 Receives completion status

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 ❶ 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 ❷, 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 ❸. 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 ❹ 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 ❺, 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) {                          ❶ Switches on string value
  case 'stop':                                          ❷ Implements stop method
    this.each(function(){
      window.clearInterval($(this).data('autoProgressbar-interval'))
    });
    break;
  case 'value':                                      Implements value method
    if (value == null) return this.progressbar('value');
    this.progressbar('value',value);
    break;
  case 'destroy':                                     Implements destroy method
    this.autoProgressbar('stop');
    this.progressbar('destroy');
    break;
  default:                                   Does nothing for unsupported strings
    break;
}
```

In this code fragment, we switch to different processing algorithms based on the string in the settings parameter ❶, 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 ❷. 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.

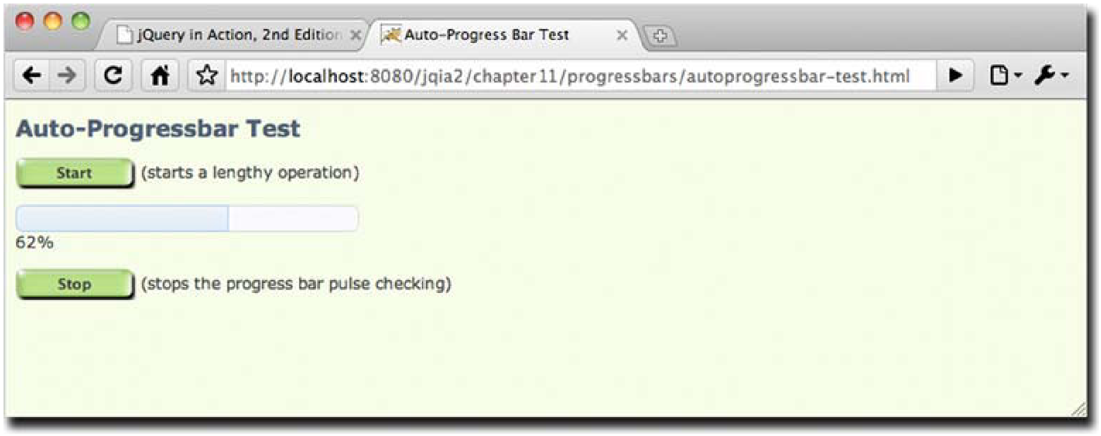
First, let's look at the markup that creates the DOM structures of note:

```
<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 `<div>` to be transformed into the progress bar.
- A `<span>` 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.


jQuery in Action, 2nd Edition
Auto-Progress Bar Test
http://localhost:8080/jqia2/chapter11/progressbars/autoprogressbar-test.html
Start
Stop

Auto-Progressbar Test

Start (starts a lengthy operation)

62%

Stop (stops the progress bar pulse checking)

Figure 11.9 The auto-progressbar is monitoring a long-running operation on the server.

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') +
     '%');
    }
  });
});
```

1 Kicks off the long-running process

2 Finalizes operation

3 Creates the monitoring progress bar

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 1. 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 3 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 ❷. 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 `<div>` 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 operations 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.

11.4.1 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.

Table 11.7 Options for the jQuery UI autocompleters

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.	✓

Table 11.7 Options for the jQuery UI autocompleters *(continued)*

Option	Description	In Lab?
`search`	(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.	✓
`select`	(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.	✓
`source`	(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.

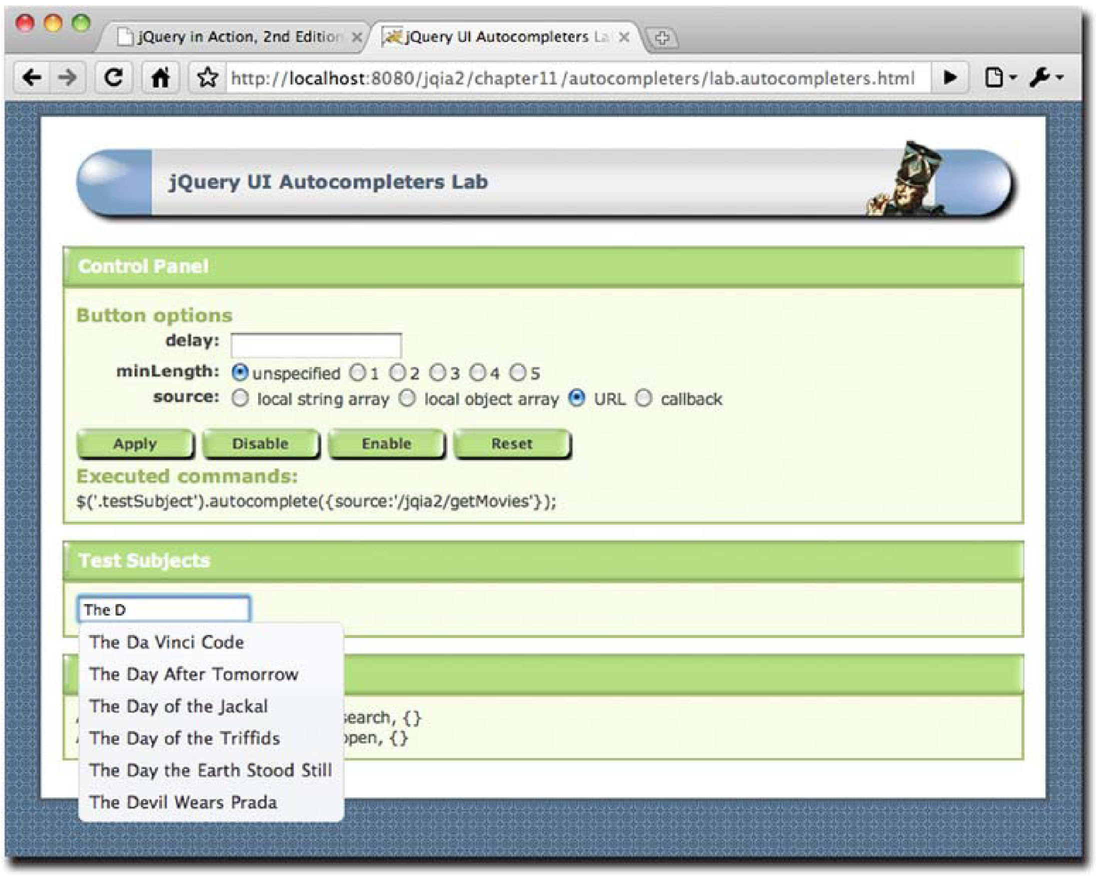
Now let's see what it takes to provide source data for this widget.

11.4.2 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.


jQuery in Action, 2nd Edition
jQuery UI Autocompleters Lab
http://localhost:8080/jqia2/chapter11/autocompleters/lab.autocompleters.html
jQuery UI Autocompleters Lab
Control Panel
Button options
Apply
Disable
Enable
Reset
Test Subjects

delay:
minLength: [✓ unspecified] 1 2 3 4 5
source: local string array local object array [✓ URL] callback

Executed commands:
$('.testSubject').autocomplete({source:'/jqia2/getMovies'});

The D

The Da Vinci Code
The Day After Tomorrow
The Day of the Jackal
The Day of the Triffids
The Day the Earth Stood Still
The Devil Wears Prada

search, {}
open, {}

Figure 11.10 The jQuery UI Autocompleters Lab shows us how a large result set can be narrowed down as more data is entered.

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 server-side 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);
  }
});
```

As with the URL variant of `source`, the result should contain only those values that are to be displayed in the autocomplete menu.

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.

11.4.3 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`	`open`	Triggered after the data has been readied and the menu is about to open.
`autocompletesearch`	`search`	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.
`autocompleteselect`	`select`	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.

11.4.4 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 `ui-autocomplete-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.

11.5 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.

11.5.1 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>`, `<div>`, and `<span>` 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.

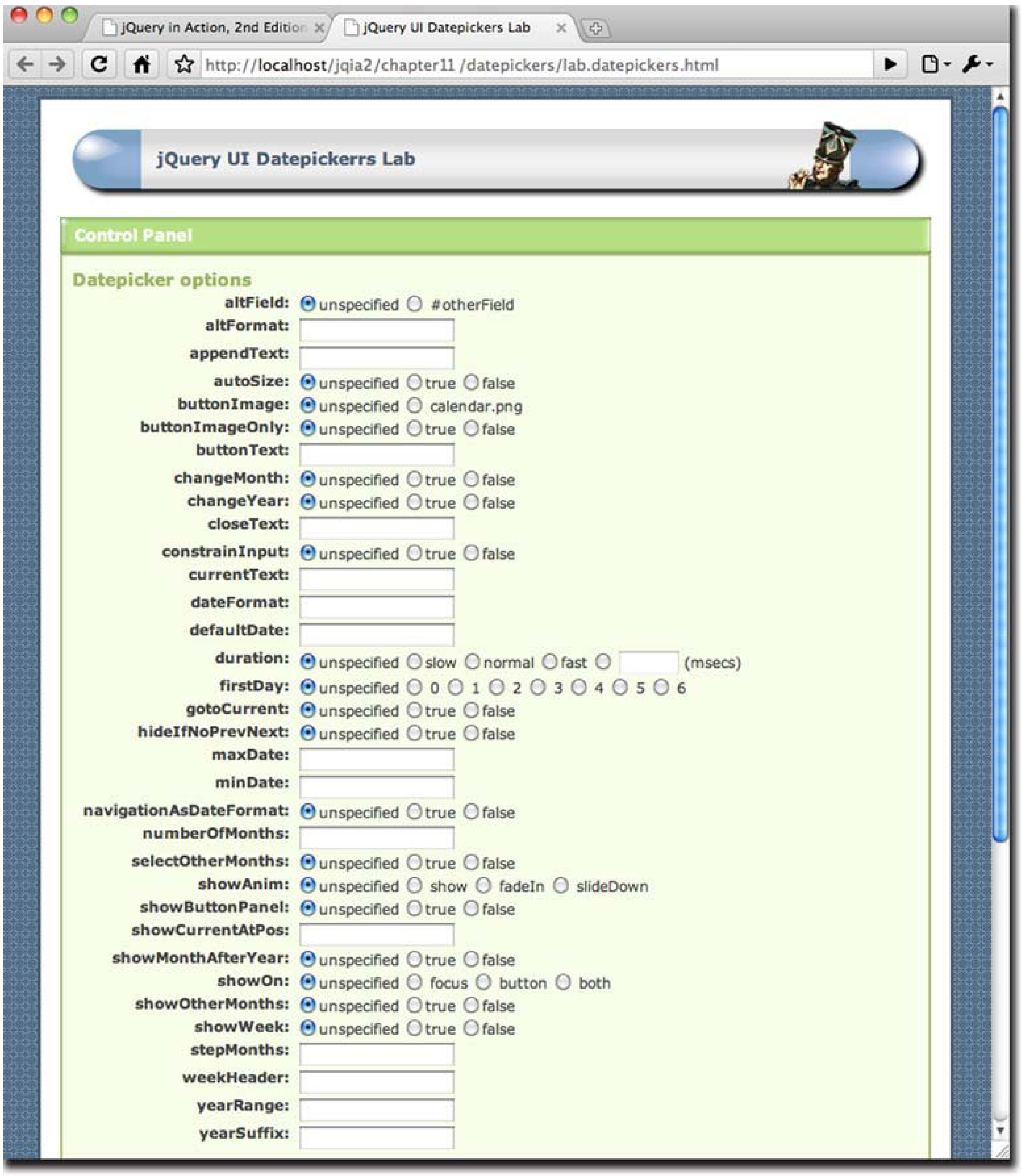
Returns

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:	

Figure 11.11 The jQuery UI Datepickers Lab helps us grasp the copious variety of options available for datepicker controls (too many to fit into one screenshot).

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.	

Table 11.9 Options for the jQuery UI datepickers ***(continued)***

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 `datapicker()` 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.	

Table 11.9 Options for the jQuery UI datepickers *(continued)*

Option	Description	In Lab?
`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 interpreted 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 triggers the datepicker when clicked. The button's appearance can be varied with the `buttonText`, `buttonImage`, and `buttonImageOnly` options. `both` causes 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.	

Table 11.9 Options for the jQuery UI datepickers *(continued)*

Option	Description	In Lab?
`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 `calculateWeek` option 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 offset 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.	✓

Still with us?

Although that may seem rather overwhelming when taken as a whole, the vast majority of options for the datepickers are used only when needed to override default values, which are usually exactly what we want. It's not uncommon to create datepickers while specifying no options at all.

11.5.2 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.

Table 11.10 Date format character patterns

Patterns	Description
`d`	Date within month without leading zeroes
`dd`	2-digit date within month with leading zeroes for values less than 10
`o`	Day of the year without leading zeroes

Table 11.10 Date format character patterns *(continued)*

Patterns	Description
`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

The datepicker defines some well-known date format patterns as constant values, as shown in table 11.11.

We'll be addressing these patterns again when we discuss the datepicker utility functions in section 11.5.4.

Now let's turn our attention to the events that datepickers trigger.

Constant	Pattern
`$.datepicker.ATOM`	`yy-mm-dd`
`$.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

11.5.3 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. The possible options are `shortYearCutoff`, `dayNames`, `dayNamesShort`, `monthNames`, and `monthNamesShort`. See table 11.9 for details of these options. If omitted, the default English names are 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.

11.6.1 Creating tabbed content

Most of the widgets we've examined so far take a simple element, such as a `<button>`, `<div>`, 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">                                  ❶ Contains tabs and
  <ul>                                               tab panels
    <li><a href="#panel1">Tab One</a></li>
    <li><a href="#panel2">Tab Two</a></li>              ❷ Defines tabs
    <li><a href="#panel3">Tab Three</a></li>
  </ul>
  <div id="panel1">
    ... content ...
  </div>
  <div id="panel2">                        ❸ Provides
    ... content ...                          panels
  </div>
  <div id="panel3">
    ... content ...
  </div>
</div>
```

This construct consists of a `<div>` element that contains the entire tabset ❶, which consists of two subsections: an unordered list (`<ul>`) containing list items (`<li>`) that will become the tabs ❷, and a set of `<div>` elements, one for each corresponding panel ❸.

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 `<div>` 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>
```

the id value of the corresponding panel would be a_third_panel. If such a panel already exists, it will be used; otherwise, it will be created.

Ajax and non-Ajax tabs can be freely mixed in a single tabset.

Once we have the base markup all squared away, we'll create the tabset with the tabs() method, applied to the outer tabset <div> element, whose syntax follows.

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

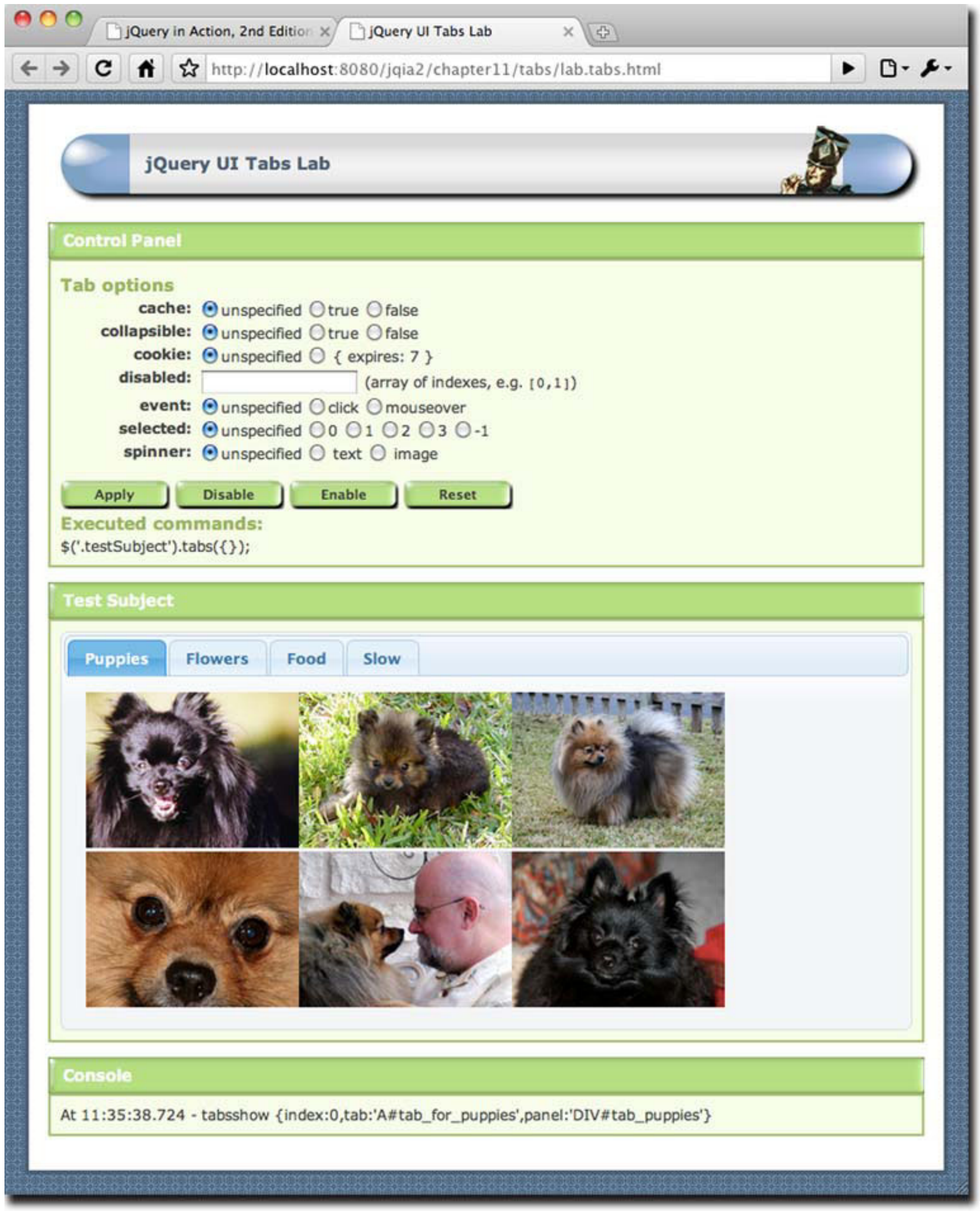
The wrapped set, except for the cases where values are returned as described above.

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.



Tab options

cache: [✓ unspecified] true false
collapsible: [✓ unspecified] true false
cookie: [✓ unspecified] { expires: 7 }
disabled: (array of indexes, e.g. [0,1])
event: [✓ unspecified] click mouseover
selected: [✓ unspecified] 0 1 2 3 -1
spinner: [✓ unspecified] text image

Executed commands:

```
$('.testSubject').tabs({});
```

```
At 11:35:38.724 - tabsshow {index:0,tab:'A#tab_for_puppies',panel:'DIV#tab_puppies'}
```

Figure 11.12 The jQuery UI Tabs Lab page shows us how tabs can be used to organize content into panels for serial display.

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 `mouseout` can 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 `"<div></div>"` 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 `"<em>Loading&#8230;</em>"`. (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 `<span>` element. For example, `<li><a href="/jqia2/lengthyTab"><span>Slow</span></a></li>`	✓
`tabTemplate`	(String) The HTML template to use when creating new tabs via the `add` method. If omitted, the default of `"<li><a href="#{href}"><span>#{label}</span></a></li>"` 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.

- *Exercise 2*—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.

11.6.2 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 canceled.
`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.

11.6.3 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; }
```

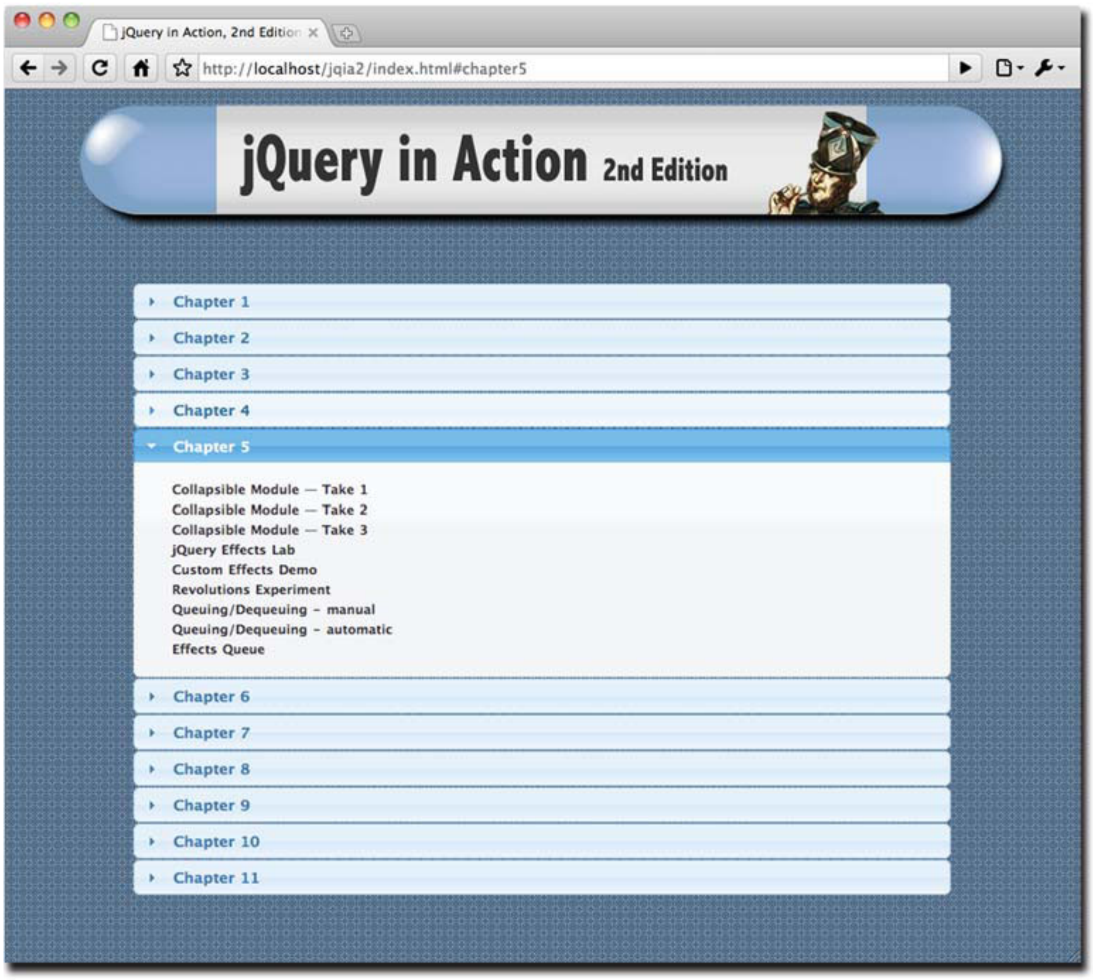
Tabs are a great and ubiquitous widget for organizing panels of related content so that users only see a single panel at a time. But what if they're a bit *too* ubiquitous and you want to achieve the same goal but with a less common look and feel?

An accordion might be just the widget for you.

11.7 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.


jQuery in Action, 2nd Edition
http://localhost/jqia2/index.html#chapter5
jQuery in Action 2nd Edition

Chapter 1
Chapter 2
Chapter 3
Chapter 4
Chapter 5
Collapsible Module — Take 1
Collapsible Module — Take 2
Collapsible Module — Take 3
jQuery Effects Lab
Custom Effects Demo
Revolutions Experiment
Queuing/Dequeuing - manual
Queuing/Dequeuing - automatic
Effects Queue
Chapter 6
Chapter 7
Chapter 8
Chapter 9
Chapter 10
Chapter 11

Figure 11.13 We used an accordion widget to organize the links to the many code examples for this book.

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.

11.7.1 *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>

</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')`
`accordion('option',optionName,value)`
`accordion('activate',index)`
`accordion('widget')`
`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`

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.	

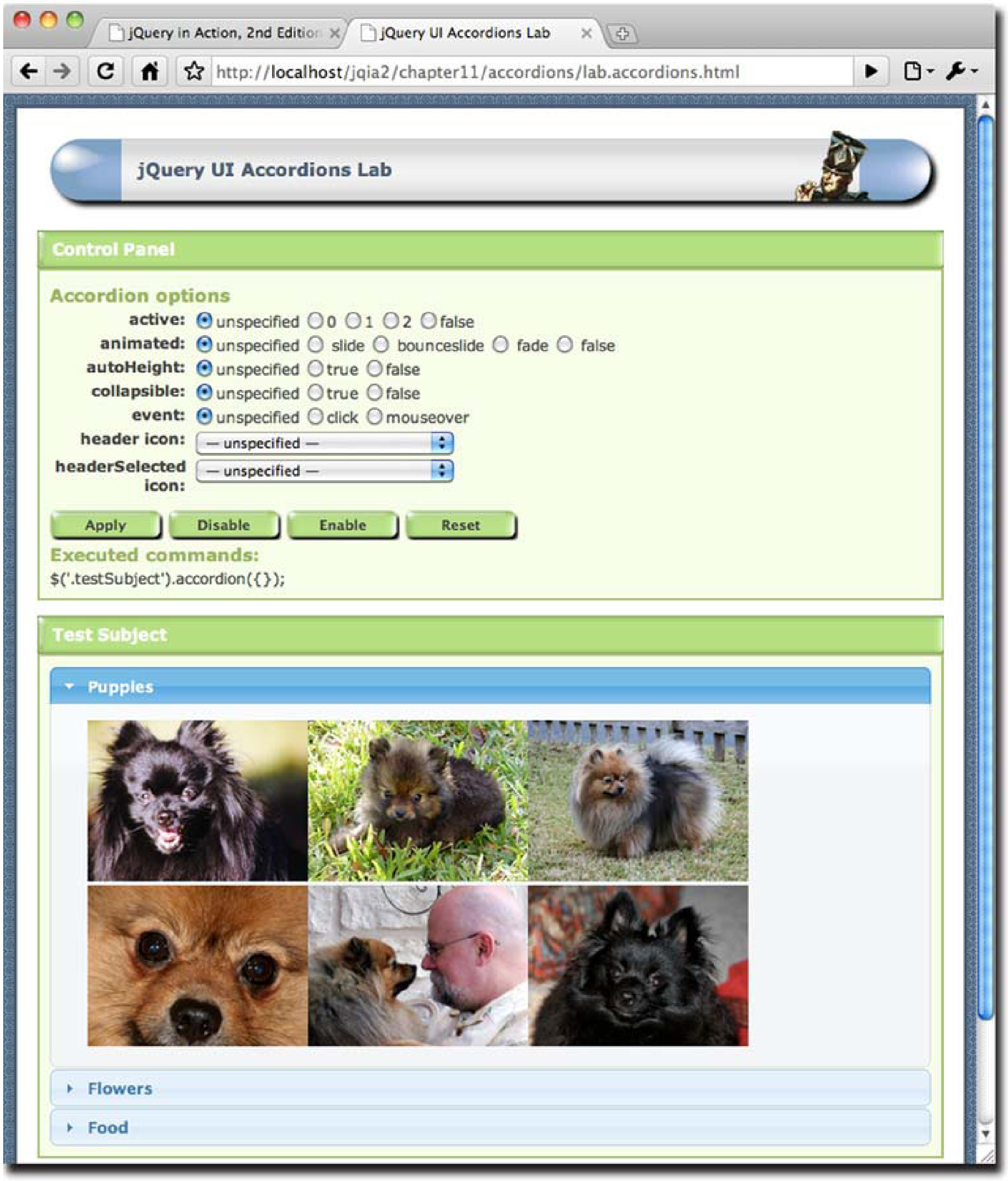
Table 11.14 Options for the jQuery UI `accordions` *(continued)*

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.



Accordion options

active: [✓ unspecified] 0 1 2 false
animated: [✓ unspecified] slide bounceslide fade false
autoHeight: [✓ unspecified] true false
collapsible: [✓ unspecified] true false
event: [✓ unspecified] click mouseover
header icon: — unspecified —
headerSelected icon: — unspecified —

Executed commands:

```
$('.testSubject').accordion({});
```

Figure 11.14 **The jQuery UI Accordions Lab shows us how we can expose serial content panels to our users in a novel fashion.**

Now we're ready to tackle the events that are triggered while an accordion is being manipulated.

11.7.2 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`	`changestart`	Triggered when the accordion is about to change.
`accordionchange`	`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.

11.7.3 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 clickable 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.

11.7.4 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.

11.8 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.

11.8.1 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 `<div>` 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 be reopened. 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.

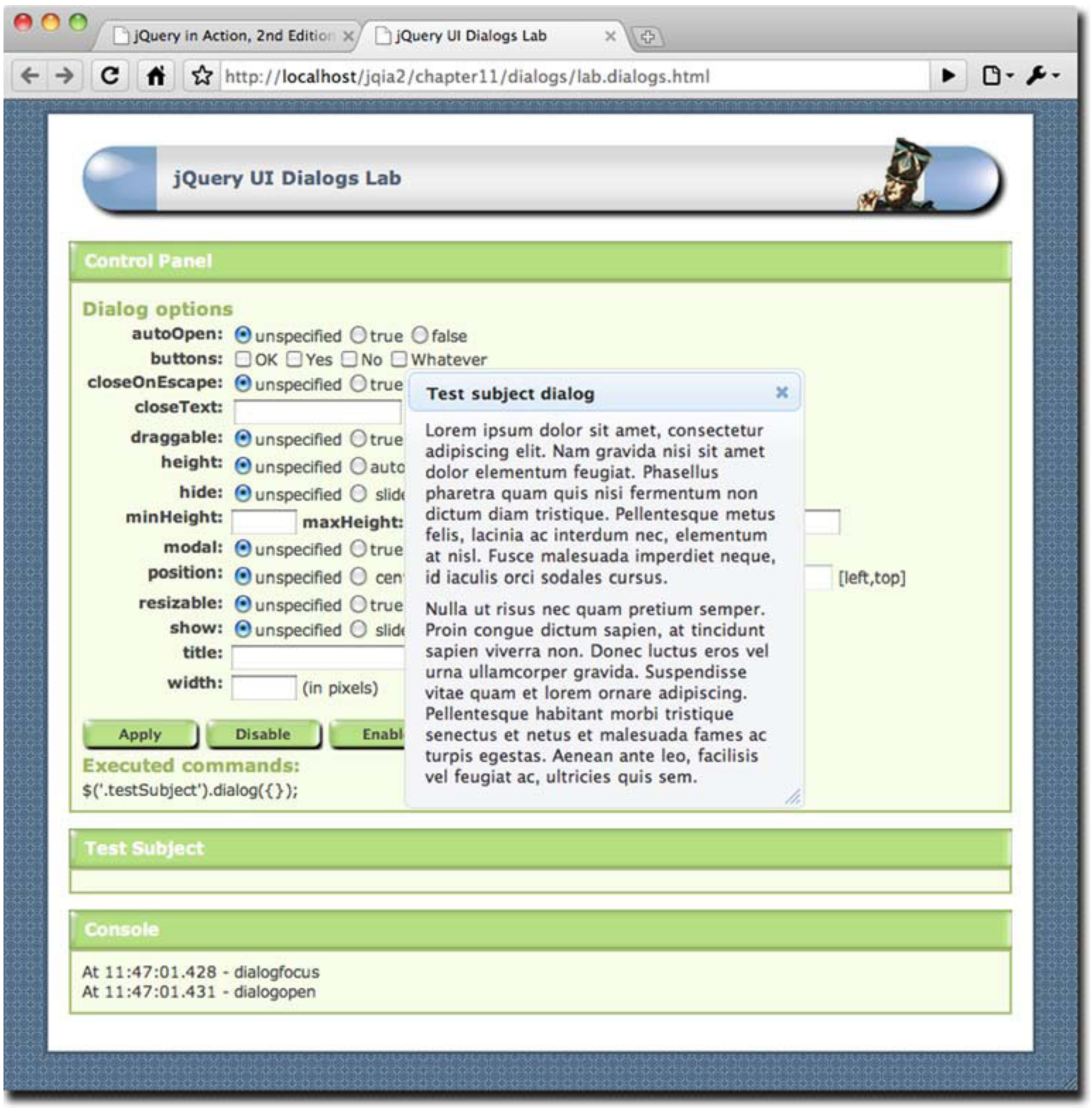
Command syntax: dialog *(continued)*

`'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.



autoOpen: [✓ unspecified] true false
buttons: ☐ OK ☐ Yes ☐ No ☐ Whatever
closeOnEscape: [✓ unspecified] true
closeText:
draggable: [✓ unspecified] true
height: [✓ unspecified] auto
hide: [✓ unspecified] slid
minHeight: maxHeight:
modal: [✓ unspecified] true
position: [✓ unspecified] cen [left,top]
resizable: [✓ unspecified] true
show: [✓ unspecified] slid
title:
width: (in pixels)

Executed commands:

```
$('.testSubject').dialog({});
```

Test subject dialog

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam gravida nisi sit amet dolor elementum feugiat. Phasellus pharetra quam quis nisi fermentum non dictum diam tristique. Pellentesque metus felis, lacinia ac interdum nec, elementum at nisl. Fusce malesuada imperdiet neque, id iaculis orci sodales cursus.

Nulla ut risus nec quam pretium semper. Proin congue dictum sapien, at tincidunt sapien viverra non. Donec luctus eros vel urna ullamcorper gravida. Suspendisse vitae quam et lorem ornare adipiscing. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean ante leo, facilisis vel feugiat ac, ultricies quis sem.

```
At 11:47:01.428 - dialogfocus
At 11:47:01.431 - dialogopen
```

Figure 11.15 **The jQuery UI Dialogs Lab lets us try out all the options that are available for jQuery UI dialog boxes.**

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

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.	✓

Table 11.16 Options for the jQuery UI dialogs *(continued)*

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

Event	Option	Description
`dialogbeforeClose`	`beforeClose`	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.
`dialogclose`	`close`	Triggered after a dialog box has closed.

Table 11.17 Events for jQuery UI dialogs *(continued)*

Event	Option	Description
`drag`	`drag`	Triggered repeatedly as a dialog box is moved about during a drag.
`dragStart`	`dragStart`	Triggered when a repositioning of the dialog box commences by dragging its title bar.
`dragStop`	`dragStop`	Triggered when a drag operation terminates.
`dialogfocus`	`focus`	Triggered when the dialog box gains focus.
`dialogopen`	`open`	Triggered when the dialog box is opened.
`resize`	`resize`	Triggered repeatedly as a dialog box is resized.
`resizeStart`	`resizeStart`	Triggered when a resize of the dialog box commences.
`resizeStop`	`resizeStop`	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 `<div>` element created to contain the entire widget, including the content and the chrome.
- `ui-dialog-titlebar`—Added to the `<div>` element created to house the title and close icon.
- `ui-dialog-title`—Added to the `<span>` 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.

11.8.4 Some dialog box tricks

Generally, dialog boxes are created from `<div>` 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 `<div>` 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!

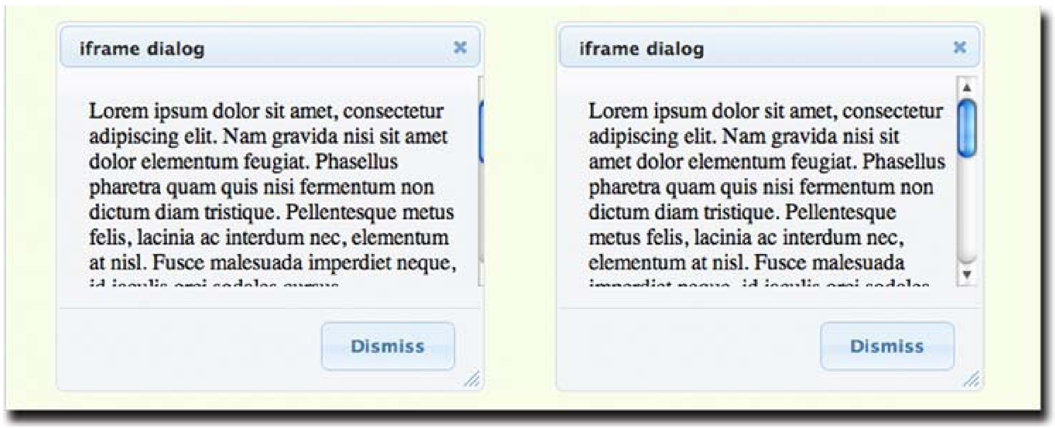
Consider this:

```
$('<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.



Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam gravida nisi sit amet dolor elementum feugiat. Phasellus pharetra quam quis nisi fermentum non dictum diam tristique. Pellentesque metus felis, lacinia ac interdum nec, elementum at nisl. Fusce malesuada imperdiet neque,

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nam gravida nisi sit amet dolor elementum feugiat. Phasellus pharetra quam quis nisi fermentum non dictum diam tristique. Pellentesque metus felis, lacinia ac interdum nec, elementum at nisl. Fusce malesuada

Figure 11.16 Our victory dance was cut short by the clipping of the scroll bar shown at left—how can we fix it as shown at right?

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.

11.9 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/.

11.10 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!