

Bear Bibeault  
Yehuda Katz

# jQuery

## IN ACTION

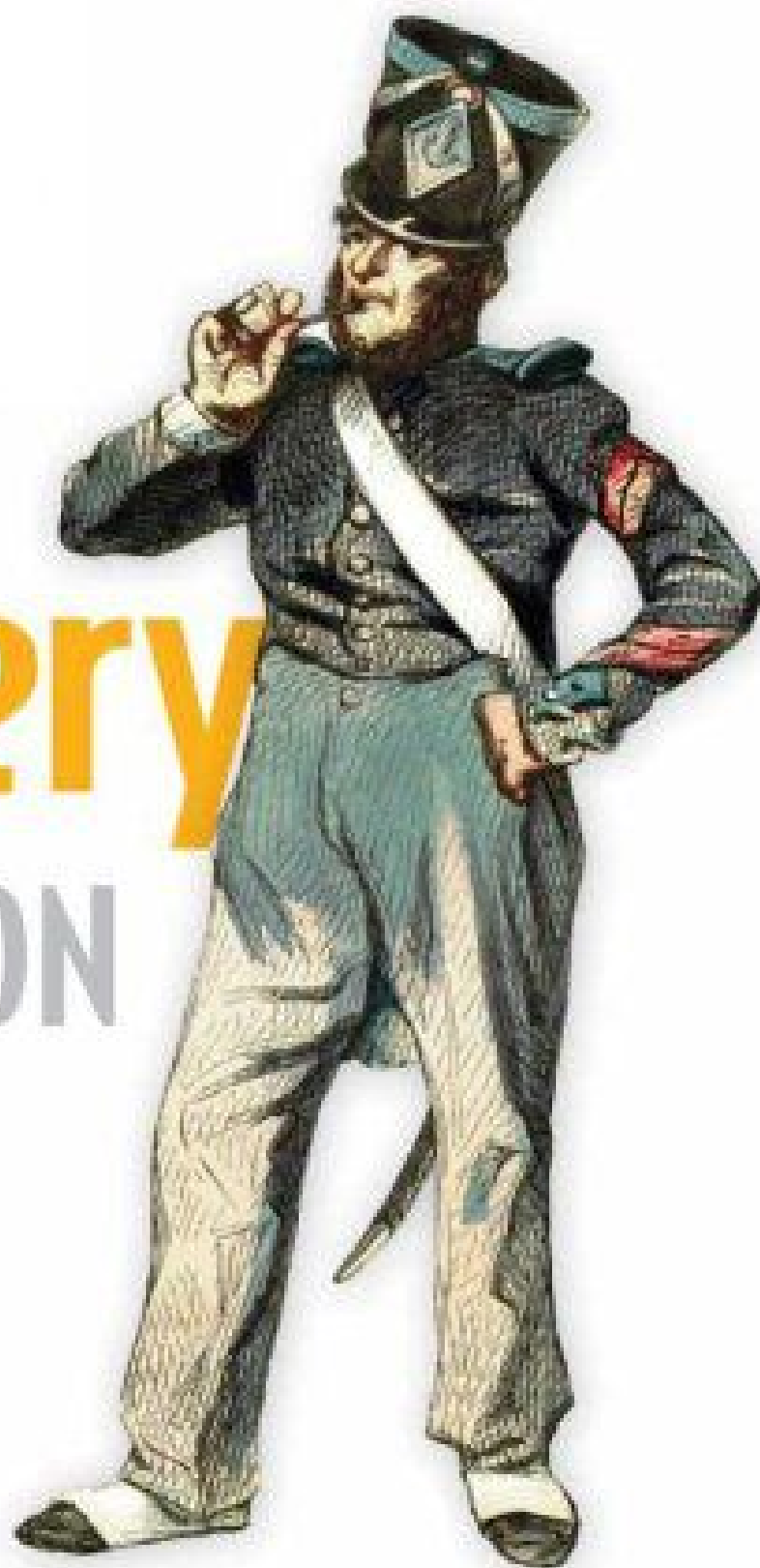
SECOND EDITION



MANNING

Copyrighted Material

Covers jQuery 1.4 and jQuery UI 1.8



Copyrighted Material

<b>Chapter 5. Energizing pages with animations and effects.....</b>	<b>1</b>
Section 5.1. Showing and hiding elements.....	2
Section 5.2. Animating the display state of elements.....	7
Section 5.3. Creating custom animations.....	17
Section 5.4. Animations and Queuing.....	22
Section 5.5. Summary.....	30

# 5

## *Energizing pages with animations and effects*

---

### ***This chapter covers***

- Showing and hiding elements without animation
- Showing and hiding elements using core animation effects
- Writing custom animations
- Controlling animation and function queuing

Browsers have come a long way since LiveScript—subsequently renamed JavaScript—was introduced to Netscape Navigator in 1995 to allow scripting of web pages.

In those early days, the capabilities afforded to page authors were severely limited, not only by the minimal APIs, but by the sluggishness of scripting engines and low-powered systems. The idea of using these limited abilities for animation and effects was laughable, and for years the only animation was through the use of animated GIF images (which were generally used poorly, making pages more annoying than usable).

My, how times have changed. Today's browser scripting engines are lightning fast, running on hardware that was unimaginable 10 years ago, and they offer a rich variety of capabilities to us as page authors.

But even though the capabilities exist in low-level operations, JavaScript has no easy-to-use animation engine, so we're on our own. Except, of course, that jQuery comes to our rescue, providing a trivially simple interface for creating all sorts of neat effects.

But before we dive into adding whiz-bang effects to our pages, we need to contemplate the question, *should we?* Like a Hollywood blockbuster that's all special effects and no plot, a page that overuses effects can elicit a very different, and negative, reaction than what we intend. Be mindful that effects should be used to *enhance* the usability of a page, not hinder it by just showing off.

With that caution in mind, let's see what jQuery has to offer.

## 5.1 Showing and hiding elements

Perhaps the most common type of dynamic effect we'll want to perform on an element, or any group of elements, is the simple act of showing or hiding them. We'll get to more fancy animations (like fading an element in or out) in a bit, but sometimes we'll want to keep it simple and pop elements into existence or make them instantly vanish!

The methods for showing and hiding elements are pretty much what we'd expect: `show()` to show the elements in a wrapped set, and `hide()` to hide them. We're going to delay presenting their formal syntax for reasons that will become clear in a bit; for now, let's concentrate on using these methods with no parameters.

As simple as these methods may seem, we should keep a few things in mind. First, jQuery hides elements by changing their `style.display` properties to `none`. If an element in the wrapped set is already hidden, it will remain hidden but still be returned for chaining. For example, suppose we have the following HTML fragment:

```
<div style="display:none;">This will start hidden</div>
<div>This will start shown</div>
```

If we apply `$("div").hide().addClass("fun")`, we'll end up with the following:

```
<div style="display:none;" class="fun">This will start hidden</div>
<div style="display:none;" class="fun">This will start shown</div>
```

Note that even though the first element was already hidden, it remains part of the matched set and takes part in the remainder of the method chain.

Second, jQuery shows objects by changing the `display` property from `none` to either `block` or `inline`. Which of these values is chosen is based upon whether a previously specified explicit value was set for the element or not. If the value was explicit, it's remembered and reverted. Otherwise it's based upon the default state of the `display` property for the target element type. For example, `<div>` elements will have their `display` property set to `block`, whereas a `<span>` element's `display` property will be set to `inline`.

Let's see about putting these methods to good use.

### 5.1.1 Implementing a collapsible “module”

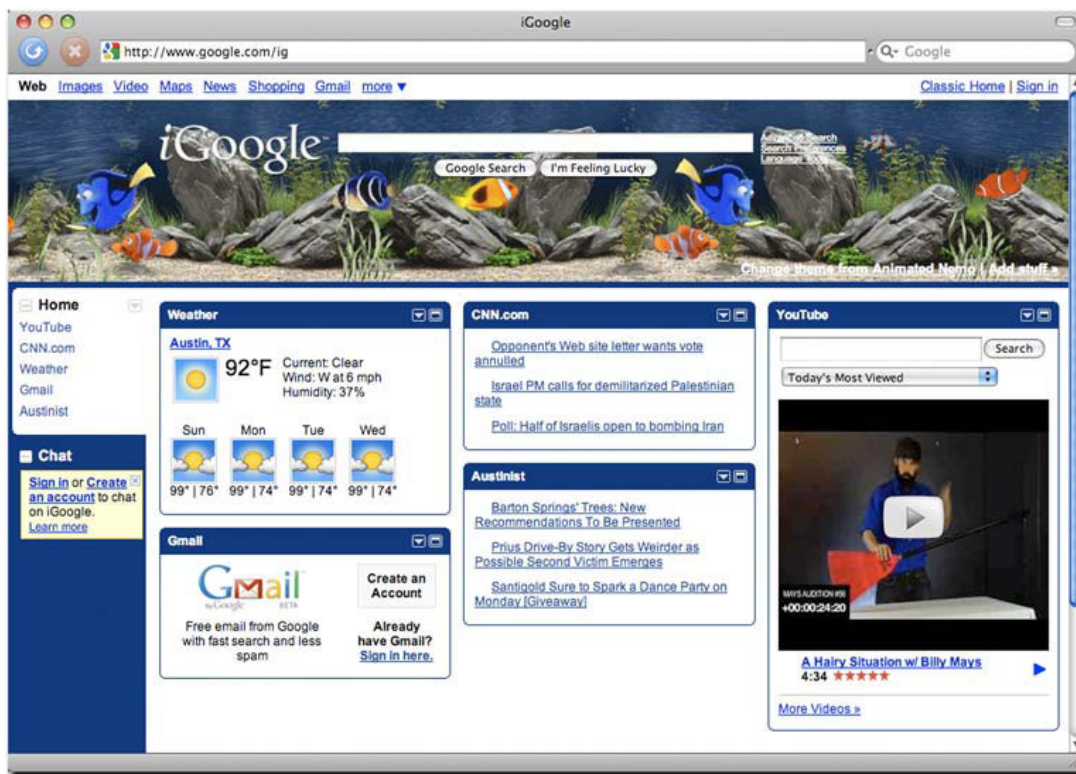
You’re no doubt familiar with sites, some of which aggregate data from other sites, that present you with various pieces of information in configurable “modules” on some sort of “dashboard” page. The iGoogle site is a good example, as shown in figure 5.1.

This site lets us configure much about how the page is presented, including moving the modules around, expanding them to full-page size, specifying configuration information, and even removing them completely. But one thing it doesn’t let us do (at least at the time of this writing) is to “roll up” a module into its caption bar so that it takes up less room, without having to remove it from the page.

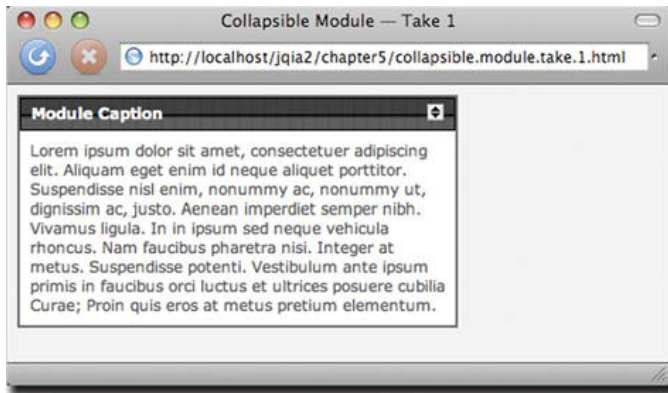
Let’s define our own dashboard modules and one-up Google by allowing users to roll up a module into its caption bar.

First, let’s take a look at what we want the module to look like in its normal and rolled-up states, shown in figures 5.2a and 5.2b respectively.

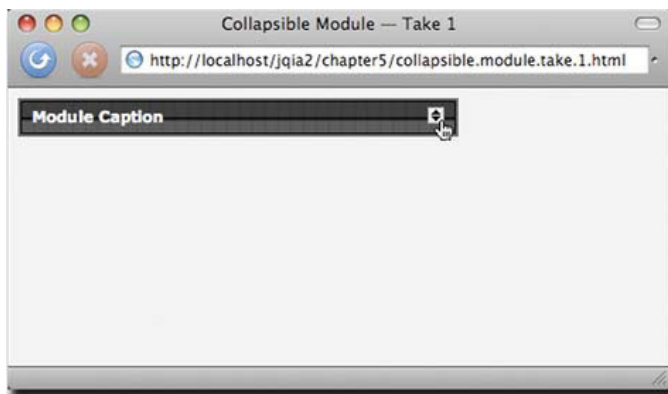
In figure 5.2a, we’ve created a module with two major sections: a caption bar, and a body. The body contains the data of the module—in this case, random “Lorem ipsum” text. The more interesting caption bar contains a caption for the module and small button that we’ll instrument to invoke the roll-up (and roll-down) functionality.



**Figure 5.1** iGoogle is an example of a site that presents aggregated information in a series of dashboard modules.



**Figure 5.2a** We'll create our own dashboard modules, which consist of two parts: a bar with a caption and roll-up button, and a body in which data can be displayed.



**Figure 5.2b** When the roll-up button is clicked, the module body disappears as if it had been rolled up into the caption bar.

Once the button is clicked, the body of the module will disappear as if it had been rolled up into the caption bar. A subsequent click will roll down the body, restoring its original appearance.

The HTML markup we've used to create the structure of our module is fairly straightforward. We've applied numerous class names to the elements both for identification as well as for CSS styling.

```
<div class="module">
  <div class="caption">
    <span>Module Caption</span>
    
  </div>
  <div class="body">
    Lorem ipsum dolor sit amet, consectetur adipiscing elit.
    Aliquam eget enim id neque aliquet porttitor. Suspendisse
    nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
    Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
    sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
    Integer at metus. Suspendisse potenti. Vestibulum ante
    ipsum primis in faucibus orci luctus et ultrices posuere
    cubilia Curae; Proin quis eros at metus pretium elementum.
  </div>
</div>
```

The entire construct is enclosed in a `<div>` element tagged with the `module` class, and the caption and body constructs are each represented as `<div>` children with the classes `caption` and `body`.

In order to give this module the roll-up behavior, we'll instrument the image in the caption with a click handler that does all the magic. And with the `hide()` and `show()` methods up our sleeves, giving the module this behavior is considerably easier than pulling a quarter out from behind someone's ear.

Let's examine the code placed into the ready handler to take care of the roll-up behavior:

```

$('div.caption img').click(function(){
  var body$ = $(this).closest('div.module').find('div.body');
  if (body$.is(':hidden')) {
    body$.show();
  }
  else {
    body$.hide();
  }
});

```

1 Instruments the button

2 Finds the related body

3 Shows the body

4 Hides the body

As would be expected, this code first establishes a click handler on the image in the caption 1.

Within the click handler, we first locate the body associated with the module. We need to find the specific instance of the module body because, remember, we may have many modules on our dashboard page, so we can't just select all elements that have the `body` class. We quickly locate the correct body element by finding the closest module container, and using it as the jQuery context, finding the body within it using the jQuery expression 2:

```
$(this).closest('div.module').find('div.body')
```

(If how this expression finds the correct element isn't clear to you, now would be a good time to review the information in chapter 2 regarding finding and selecting elements.)

Once the body is located, it becomes a simple matter of determining whether the body is hidden or shown (the jQuery `is()` method comes in mighty handy here), and either showing or hiding it as appropriate using the `show()` 3 or `hide()` 4 method.

**NOTE** With this code example, we've introduced a convention that many people use when storing references to a wrapped set within a variable: that of using the `$` character within the variable name. Some may use the `$` as a prefix and some as a suffix (as we have done here—if you think of the `$` as representing the word “wrapper,” the variable name `body$` can be read as “body wrapper”). In either case, it's a handy way to remember that the variable contains a reference to a wrapped set rather than an element or other type of object.

The full code for this page can be found in file `chapter5/collapsible.module.take.1.html` and is shown in listing 5.1. (If you surmise that the “take 1” part of this filename indicates that we'll be revisiting this example, you're right!)

**Listing 5.1 The first implementation of our collapsible module**

```

<!DOCTYPE html>
<html>
  <head>
    <title>Collapsible Module &mdash; Take 1</title>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css" />
    <link rel="stylesheet" type="text/css" href="module.css" />
    <script type="text/javascript" src="../../scripts/jquery-1.4.js"></script>
    <script type="text/javascript">
      $(function() {

        $('div.caption img').click(function(){
          var body$ = $(this).closest('div.module').find('div.body');
          if (body$.is(':hidden')) {
            body$.show();
          }
          else {
            body$.hide();
          }
        });
      });
    </script>
  </head>

  <body class="plain">

    <div class="module">
      <div class="caption">
        <span>Module Caption</span>
        
      </div>
      <div class="body">
        Lorem ipsum dolor sit amet, consectetur adipiscing elit.
        Aliquam eget enim id neque aliquet porttitor. Suspendisse
        nisl enim, nonummy ac, nonummy ut, dignissim ac, justo.
        Aenean imperdiet semper nibh. Vivamus ligula. In in ipsum
        sed neque vehicula rhoncus. Nam faucibus pharetra nisi.
        Integer at metus. Suspendisse potenti. Vestibulum ante
        ipsum primis in faucibus orci luctus et ultrices posuere
        cubilia Curae; Proin quis eros at metus pretium elementum.
      </div>
    </div>

  </body>
</html>

```

That wasn't difficult at all, was it? But as it turns out, it can be even easier!

### 5.1.2 Toggling the display state of elements

Toggling the display state of elements between revealed and hidden—as we did for the collapsible module example—is such a common occurrence that jQuery defines a method named `toggle()` that makes it even easier.



Let's apply this method to the collapsible module and see how it helps to simplify the code of listing 5.1. Listing 5.2 shows only the ready handler for the refactored page (no other changes are necessary) with the changes highlighted in bold. The complete page code can be found in file `chapter5/collapsible.module.take.2.html`.

**Listing 5.2 The collapsible module code, simplified with `toggle()`**

```
$(function() {  
    $('div.caption img').click(function(){  
        $(this).closest('div.module').find('div.body').toggle();  
    });  
});
```

Note that we no longer need the conditional statement to determine whether to hide or show the module body; `toggle()` takes care of swapping the displayed state on our behalf. This allows us to simplify the code quite a bit, and the need to store the body reference in a variable simply vanishes.

Instantaneously making elements appear and disappear is handy, but sometimes we want the transition to be less abrupt. Let's see what's available for that.

## 5.2 Animating the display state of elements

Human cognitive ability being what it is, making items pop into and out of existence instantaneously can be jarring to us. If we blink at the wrong moment, we could miss the transition, leaving us to wonder, "What just happened?"

Gradual transitions of a *short* duration help us know what's changing and *how* we got from one state to the other—and that's where the jQuery core effects come in. There are three sets of effect types:

- Show and hide (there's a bit more to these methods than we let on in section 5.1)
- Fade in and fade out
- Slide down and slide up

Let's look more closely at each of these effect sets.

### 5.2.1 Showing and hiding elements gradually

The `show()`, `hide()`, and `toggle()` methods are a tad more complex than we led you to believe in the previous section. When called with no parameters, these methods effect a simple manipulation of the display state of the wrapped elements, causing them to instantaneously be revealed or hidden from the display. But when passed parameters, these effects can be animated so that the changes in display status of the affected elements take place over a period of time.

With that, we're now ready to look at the full syntaxes of these methods.

**Method syntax: hide****hide(speed, callback)**

Causes the elements in the wrapped set to become hidden. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to `none`. If a `speed` parameter is provided, the elements are hidden over a period of time by adjusting their width, height, and opacity downward to zero, at which time their `display` style property value is set to `none` to remove them from the display.

An optional callback can be specified, which is invoked when the animation is complete.

**Parameters**

<code>speed</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, no animation takes place, and the elements are immediately removed from the display.
<code>callback</code>	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. The callback is fired for each element that undergoes animation.

**Returns**

The wrapped set.

**Method syntax: show****show(speed, callback)**

Causes any hidden elements in the wrapped set to be revealed. If called with no parameters, the operation takes place instantaneously by setting the `display` style property value of the elements to an appropriate setting (one of `block` or `inline`).

If a `speed` parameter is provided, the elements are revealed over a specified duration by adjusting their width, height, and opacity upward to full size and opacity.

An optional callback can be specified that's invoked when the animation is complete.

**Parameters**

<code>speed</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, no animation takes place and the elements are immediately revealed in the display.
<code>callback</code>	(Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. The callback is fired for each element that undergoes animation.

**Returns**

The wrapped set.

**Method syntax: toggle****toggle(speed, callback)**

Performs `show()` on any hidden wrapped elements and `hide()` on any non-hidden wrapped elements. See the syntax description of those methods for their semantics.

**Parameters**

<code>speed</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, no animation takes place.
<code>callback</code>	(Function) An optional function invoked when the animation is complete. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. The callback is fired for each element that undergoes animation.

**Returns**

The wrapped set.

We can exert a bit more control over the toggling process with another variant of the `toggle()` method:

**Method syntax: toggle****toggle(condition)**

Shows or hides the matched elements based upon the evaluation of the passed condition. If `true`, the elements are shown; otherwise, they're hidden.

**Parameters**

<code>condition</code>	(Boolean) Determines whether elements are shown (if <code>true</code> ), or hidden (if <code>false</code> ).
------------------------	--

**Returns**

The wrapped set.

Let's do a third take on the collapsible module, animating the opening and closing of the sections.

Given the previous information, you'd think that the only change we'd need to make to the code in listing 5.2 would be to change the call to the `toggle()` method to `toggle('slow')`

And you'd be right.

But not so fast! Because that was just *too* easy, let's take the opportunity to also add some whizz-bang to the module.

Let's say that, to give the user an unmistakable visual clue, we want the module's caption to display a different background when it's in its rolled-up state. We could make the change before firing off the animation, but it'd be much more suave to wait until the animation is finished.

We can't just make the call right after the animation method call because animations *don't block*. The statements following the animated method call would execute immediately, probably even before the animation has had a chance to commence.

Rather, that's where the callback that we can register with the `toggle()` method comes in.

The approach that we'll take is that, after the animation is complete, we'll add a class name to the module to indicate that it's rolled up, and remove the class name when it isn't rolled up. CSS rules will take care of the rest.

Your initial thoughts might have led you to think of using `css()` to add a background style property directly to the caption, but why use a sledgehammer when we have a scalpel?

The "normal" CSS rule for the module caption (found in file `chapter5/module.css`) is as follows:

```
div.module div.caption {
  background: black url('module.caption.backg.png');
  ...
}
```

We've also added another rule:

```
div.module.rolledup div.caption {
  background: black url('module.caption.backg.rolledup.png');
}
```

This second rule causes the background image of the caption to change whenever its parent module possesses the `rolledup` class. So, in order to effect the change, all we have to do is add or remove the `rolledup` class to the module at the appropriate points.

Listing 5.3 shows the new ready handler code that makes that happen.

### Listing 5.3 Animated version of the module, now with a magically changing caption!

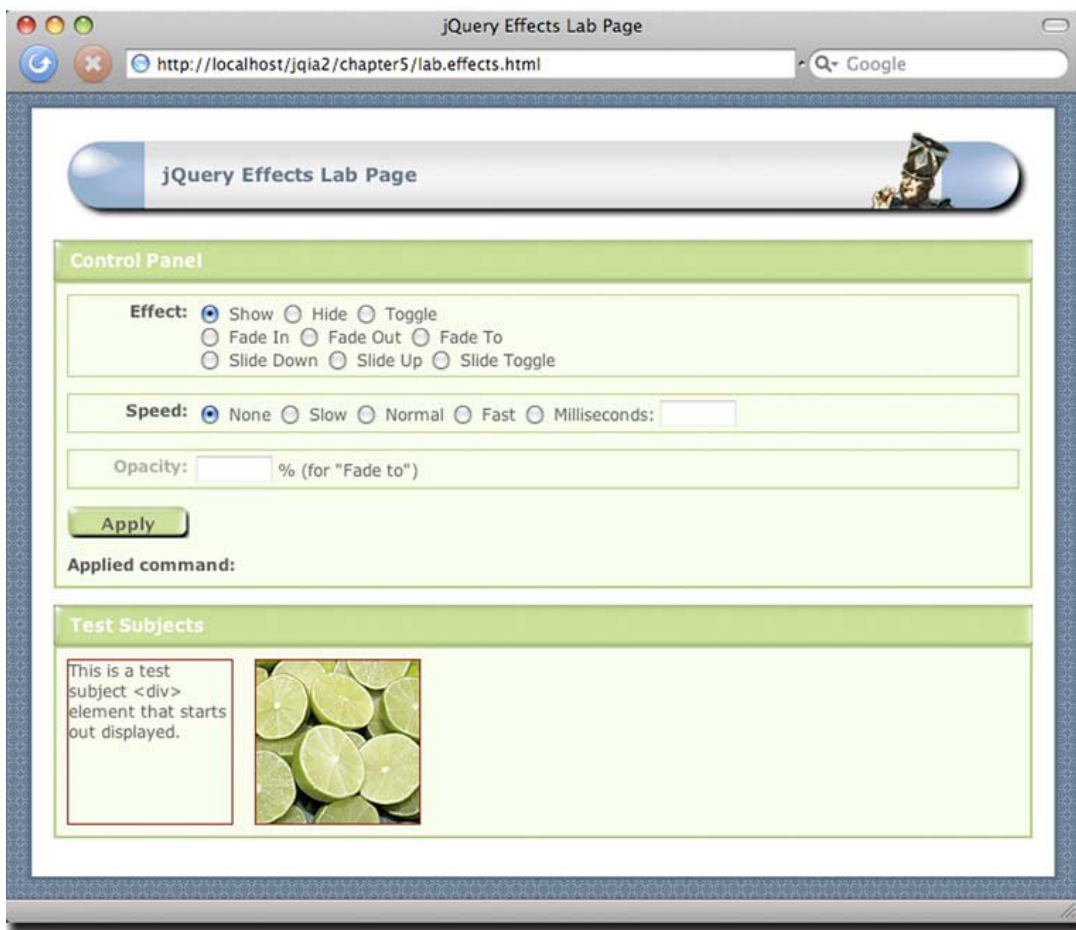
```
$(function() {
  $('div.caption img').click(function(){
    $(this).closest('div.module').find('div.body')
      .toggle('slow',function(){
        $(this).closest('div.module')
          .toggleClass('rolledup',$(this).is(':hidden'));
      });
  });
});
```

The page with these changes can be found in file `chapter5/collapsible.list.take.3.html`.

Knowing how much people like us love to tinker, we've set up a handy tool that we'll use to further examine the operation of these and the remaining effects methods.

### INTRODUCING THE JQUERY EFFECTS LAB PAGE

Back in chapter 2, we introduced the concept of lab pages to help us experiment with using jQuery selectors. For this chapter, we've set up a jQuery Effects Lab Page for exploring the operation of the jQuery effects in file `chapter5/lab.effects.html`.



**Figure 5.3** The initial state of the jQuery Effects Lab Page, which will help us examine the operation of the jQuery effects methods.

Loading this page into your browser results in the display of figure 5.3.



This Lab consists of two main panels: a control panel in which we'll specify which effect will be applied, and one that contains four test subject elements upon which the effects will act.

"Are they daft?" you might be thinking. "There are only two test subjects."

No, your authors haven't lost it yet. There *are* four elements, but two of them (another `<div>` with text and another image) are initially hidden.



Let's use this page to demonstrate the operations of the methods we've discussed to this point. Display the page in your browser, and follow along with the ensuing exercises:

- **Exercise 1**—With the controls left as is after the initial page load, click the Apply button. This will execute a `show()` method with no parameters. The expression that was applied is displayed below the Apply button for your information. Note how the two initially hidden test subject elements appear instantly. If you're

wondering why the belt image on the far right appears a bit faded, its opacity has been purposefully set to 50 percent.

- *Exercise 2*—Select the Hide radio button, and click Apply to execute a parameter-less `hide()` method. All of the test subjects immediately vanish. Take special notice that the pane in which they resided has tightened up. This indicates that the elements have been completely removed from the display rather than merely made invisible.

**NOTE** When we say that an element has been *removed from the display* (here, and in the remainder of our discussion about effects), we mean that the element is no longer being taken into account by the browser's layout manager by setting its CSS `display` style property to `none`. It doesn't mean that the element has been removed from the DOM tree; none of the effects will ever cause an element to be removed from the DOM.

- *Exercise 3*—Select the Toggle radio button, and click Apply. Click Apply again. And again. You'll note that each subsequent execution of `toggle()` flips the presence of the test subjects.
- *Exercise 4*—Reload the page to reset everything to the initial conditions (in Firefox and other Gecko browsers, set focus to the address bar and press the Enter key—simply clicking the reload button won't reset the form elements). Select Toggle, and click Apply. Note how the two initially visible subjects vanish and the two that were hidden appear. This demonstrates that the `toggle()` method applies individually to each wrapped element, revealing the ones that are hidden and hiding those that aren't.
- *Exercise 5*—In this exercise, we'll move into the realm of animation. Refresh the page, leave Show selected, and select Slow for the Speed setting. Click Apply, and carefully watch the test subjects. The two hidden elements, rather than popping into existence, gradually grow from their upper-left corners. If you want to really see what's going on, refresh the page, select Milliseconds for the speed and enter 10000 for the speed value. This will extend the duration of the effect to 10 (excruciating) seconds and give you plenty of time to observe the behavior of the effect.
- *Exercise 6*—Choosing various combinations of Show, Hide, and Toggle, as well as various speeds, experiment with these effects until you feel you have a good handle on how they operate.

Armed with the jQuery Effects Lab Page, and the knowledge of how this first set of effects operates, let's take a look at the next set of effects.

### 5.2.2 Fading elements into and out of existence

If you watched the operation of the `show()` and `hide()` effects carefully, you will have noted that they scaled the size of the elements (either up or down as appropriate) *and* adjusted the opacity of the elements as they grew or shrank. The next set of effects, `fadeIn()` and `fadeOut()`, only affects the opacity of the elements.

Other than the lack of scaling, these methods work in a fashion similar to the animated forms of `show()` and `hide()`. The syntaxes of these methods are as follows:

#### Method syntax: **fadeIn**

##### **fadeIn(speed, callback)**

Causes any matched elements that are hidden to be shown by gradually changing their opacity to their natural value. This value is either the opacity originally applied to the element, or 100 percent. The duration of the change in opacity is determined by the `speed` parameter. Only hidden elements are affected.

##### **Parameters**

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, the default is “normal”.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

##### **Returns**

The wrapped set.

#### Method syntax: **fadeOut**

##### **fadeOut(speed, callback)**

Causes any matched elements that aren’t hidden to be removed from the display by gradually changing their opacity to 0 percent and then removing the element from the display. The duration of the change in opacity is determined by the `speed` parameter. Only displayed elements are affected.

##### **Parameters**

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, the default is “normal”.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

##### **Returns**

The wrapped set.



Let’s have some more fun with the jQuery Effects Lab Page. Display the Lab, and run through a set of exercises similar to those in the previous section but using the Fade In and Fade Out selections (don’t worry about Fade To for now; we’ll attend to that soon enough).

It’s important to note that when the opacity of an element is adjusted, the jQuery `hide()`, `show()`, `fadeIn()`, and `fadeOut()` effects remember the original opacity of an element and honor its value. In the Lab, we purposefully set the initial opacity of the belt image at the far right to 50 percent before hiding it. Throughout all the opacity changes that take place when applying the jQuery effects, this original value is never stomped on.



Run through additional exercises in the Lab until you're convinced that this is so and are comfortable with the operation of the fade effects.

Another effect that jQuery provides is via the `fadeTo()` method. This effect adjusts the opacity of the elements like the previously examined fade effects, but it never removes the elements from the display. Before we start playing with `fadeTo()` in the Lab, here's its syntax.

#### Method syntax: `fadeTo`

##### **`fadeTo(speed,opacity,callback)`**

Gradually adjusts the opacity of the wrapped elements from their current settings to the new setting specified by `opacity`.

##### **Parameters**

<code>speed</code>	(Number String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted, the default is "normal".
<code>opacity</code>	(Number) The target opacity to which the elements will be adjusted, specified as a value from 0.0 to 1.0.
<code>callback</code>	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. This callback is fired individually for each animated element.

##### **Returns**

The wrapped set.

Unlike the other effects that adjust opacity while hiding or revealing elements, `fadeTo()` doesn't remember the original opacity of an element. This makes sense because the whole purpose of this effect is to explicitly change the opacity to a specific value.



Bring up the Lab page, and cause all elements to be revealed (you should know how by now). Then work through the following exercises:

- *Exercise 1*—Select Fade To and a speed value slow enough for you to observe the behavior; 4000 milliseconds is a good choice. Now set the Opacity field (which expects a percentage value between 0 and 100, converted to 0.0 through 1.0 when passed to the method) to 10, and click Apply. The test subjects will fade to 10 percent opacity over the course of four seconds.
- *Exercise 2*—Set the opacity to 100, and click Apply. All elements, including the initially semitransparent belt image, are adjusted to full opaqueness.
- *Exercise 3*—Set the opacity to 0, and click Apply. All elements fade away to invisibility, but note that once they've vanished, the enclosing module doesn't tighten up. Unlike the `fadeOut()` effect, `fadeTo()` never removes the elements from the display, even when they're fully invisible.

Continue experimenting with the Fade To effect until you've mastered its workings. Then we'll be ready to move on to the next set of effects.



### 5.2.3 *Sliding elements up and down*

Another set of effects that hide or show elements—`slideDown()` and `slideUp()`—also works in a similar manner to the `hide()` and `show()` effects, except that the elements appear to slide down from their tops when being revealed and to slide up into their tops when being hidden.

As with `hide()` and `show()`, the slide effects have a related method that will toggle the elements between hidden and revealed: `slideToggle()`. The by-now-familiar syntaxes for these methods follow.

#### Method syntax: `slideDown`

##### **`slideDown(speed, callback)`**

Causes any matched elements that are hidden to be shown by gradually increasing their vertical size. Only hidden elements are affected.

##### **Parameters**

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, the default is “normal”.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

##### **Returns**

The wrapped set.

#### Method syntax: `slideUp`

##### **`slideUp(speed, callback)`**

Causes any matched elements that are displayed to be removed from the display by gradually decreasing their vertical size.

##### **Parameters**

- `speed` (Number|String) Specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, the default is “normal”.
- `callback` (Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context (`this`) is set to the element that was animated. This callback is fired individually for each animated element.

##### **Returns**

The wrapped set.

**Method syntax: slideToggle****slideToggle(speed, callback)**

Performs `slideDown()` on any hidden wrapped elements and `slideUp()` on any displayed wrapped elements. See the syntax description of those methods for their semantics.

**Parameters**

<code>speed</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: “slow”, “normal”, or “fast”. If omitted, the default is “normal”.
<code>callback</code>	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. This callback is fired individually for each animated element.

**Returns**

The wrapped set.



Except for the manner in which the elements are revealed and hidden, these effects act similarly to the other show and hide effects. Convince yourself of this by displaying the jQuery Effects Lab Page and running through exercises like those we applied using the other effects.

**5.2.4 Stopping animations**

We may have a reason now and again to stop an animation once it has started. This could be because a user event dictates that something else should occur or because we want to start a completely new animation. The `stop()` command will achieve this for us.

**Command syntax: stop****stop(clearQueue, gotoEnd)**

Halts any animations that are currently in progress for the elements in the matched set.

**Parameters**

<code>clearQueue</code>	(Boolean) If specified and set to <code>true</code> , not only stops the current animation, but any other animations waiting in the animation queue. (Animation queue? We'll get to that shortly ...)
<code>gotoEnd</code>	(Boolean) If specified and set to <code>true</code> , advances the current animation to its logical end (as opposed to merely stopping it).

**Returns**

The wrapped set.

Note that any changes that have already taken place for any animated elements will remain in effect. If we want to restore the elements to their original states, it's our responsibility to change the CSS values back to their starting values using the `css()` method or similar methods.

By the way, there's also a global flag that we can use to completely disable all animations. Setting the flag `jQuery.fx.off` to `true` will cause all effects to take place immediately without animation. We'll cover this flag more formally in chapter 6 with the other jQuery flags.

Now that we've seen the effects built into core jQuery, let's investigate writing our own!

### 5.3 *Creating custom animations*

The number of core effects supplied with jQuery is purposefully kept small, in order to keep jQuery's core footprint to a minimum, with the expectation that page authors would use plugins (including jQuery UI which we will begin to explore in chapter 9) to add more animations at their discretion. It's also a surprisingly simple matter to write our own animations.

jQuery publishes the `animate()` wrapper method, which allows us to apply our own custom animated effects to the elements of a wrapped set. Let's take a look at its syntax.

#### Method syntax: `animate`

**`animate(properties, duration, easing, callback)`**

**`animate(properties, options)`**

Applies an animation, as specified by the `properties` and `easing` parameters, to all members of the wrapped set. An optional callback function can be specified that's invoked when the animation is complete. An alternative format specifies a set of `options` in addition to the `properties`.

#### Parameters

<code>properties</code>	(Object) An object hash that specifies the values that supported CSS styles should reach at the end of the animation. The animation takes place by adjusting the values of the style properties from the current value for an element to the value specified in this object hash. (Be sure to use camel case when specifying multiword properties.)
<code>duration</code>	(Number String) Optionally specifies the duration of the effect as a number of milliseconds or as one of the predefined strings: "slow", "normal", or "fast". If omitted or specified as 0, no animation takes place and the elements' specified <code>properties</code> are immediately, and synchronously, set to the target values.
<code>easing</code>	(String) The optional name of a function to perform easing of the animation. Easing functions must be registered by name and are often provided by plugins. Core jQuery supplies two easing functions registered as "linear" and "swing". (See chapter 9 for the list of easing functions provided by jQuery UI.)
<code>callback</code>	(Function) An optional function invoked when the animation completes. No parameters are passed to this function, but the function context ( <code>this</code> ) is set to the element that was animated. This callback is fired individually for each animated element.
<code>options</code>	(Object) Specifies the animation parameter values using an object hash. The supported properties are as follows: <ul style="list-style-type: none"> <li>- <code>duration</code>—See previous description of <code>duration</code> parameter.</li> <li>- <code>easing</code>—See previous description of <code>easing</code> parameter.</li> <li>- <code>complete</code>—Function invoked when the animation completes.</li> <li>- <code>queue</code>—If <code>false</code>, the animation isn't queued and begins running immediately.</li> <li>- <code>step</code>—A callback function called at each step of the animation. This callback is passed the step index and an internal effects object (that doesn't contain much of interest to us as page authors). The function context is set to the element under animation.</li> </ul>

#### Returns

The wrapped set.

We can create custom animations by supplying a set of CSS style properties and target values that those properties will converge towards as the animation progresses. Animations start with an element's original style value and proceed by adjusting that style value in the direction of the target value. The intermediate values that the style achieves during the effect (automatically handled by the animation engine) are determined by the duration of the animation and the easing function.

The specified target values can be absolute values, or we can specify relative values from the starting point. To specify relative values, prefix the value with `+=` or `-=` to indicate relative target values in the positive or negative direction, respectively.

The term *easing* is used to describe the manner in which the processing and pace of the frames of the animation are handled. By using some fancy math on the duration of the animation and current time position, some interesting variations to the effects are possible. The subject of writing easing functions is a complex, niche topic that's usually only of interest to the most hard-core of plugin authors; we're not going to delve into the subject of custom easing functions in this book. We'll be taking a look at a lot more easing functions in chapter 9 when we examine jQuery UI.

By default, animations are added to a queue for execution (much more on that coming up); applying multiple animations to an object will cause them to run serially. If you'd like to run animations in parallel, set the `queue` option to `false`.

The list of CSS style properties that can be animated is limited to those that accept numeric values for which there is a logical progression from a start value to a target value. This numeric restriction is completely understandable—how would we envision the logical progress from a source value to an end value for a non-numeric property such as `background-image`? For values that represent dimensions, jQuery assumes the default unit of pixels, but we can also specify `em` units or percentages by including the `em` or `%` suffixes.

Frequently animated style properties include `top`, `left`, `width`, `height`, and `opacity`. But if it makes sense for the effect we want to achieve, numeric style properties such as font size, margin, padding, and border dimensions can also be animated.

**NOTE** jQuery UI adds the ability to animate CSS properties that specify a color value. We'll learn all about that when we discuss jQuery UI effects in chapter 9.

In addition to specific values for the target properties, we can also specify one of the strings `"hide"`, `"show"`, or `"toggle"`; jQuery will compute the end value as appropriate to the specification of the string. Using `"hide"` for the `opacity` property, for example, will result in the opacity of an element being reduced to 0. Using any of these special strings has the added effect of automatically revealing or removing the element from the display (like the `hide()` and `show()` methods), and it should be noted that `"toggle"` remembers the initial state so that it can be restored on a subsequent `"toggle"`.

Did you notice that when we introduced the core animations that there was no toggling method for the fade effects? That's easily solved using `animate()` and `"toggle"` to create a simple custom animation as follows:

```
$('.animateMe').animate({opacity:'toggle'}, 'slow');
```

Taking this to the next logical step—creating a wrapper function—could be coded as follows:

```
$.fn.fadeToggle = function(speed){
    return this.animate({opacity:'toggle'},speed);
};
```

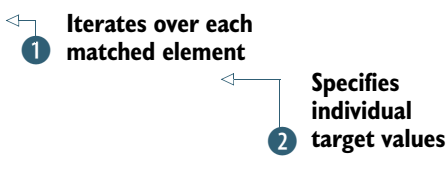
Now, let's try our hand at writing a few more custom animations.

### 5.3.1 A custom scale animation

Consider a simple *scale* animation in which we want to adjust the size of the elements to twice their original dimensions. We'd write such an animation as shown in listing 5.4.

**Listing 5.4 A custom scale animation**

```
$('.animateMe').each(function(){
    $(this).animate({
        width: $(this).width() * 2,
        height: $(this).height() * 2
    },
    2000
    );
});
```



To implement this animation, we iterate over all the elements in the wrapped set via `each()` to apply the animation individually to each matched element ❶. This is important because the property values that we need to specify for each element are based upon the individual dimensions for that element ❷. If we always knew that we'd be animating a single element (such as if we were using an `id` selector) or applying the exact same set of values to each element, we could dispense with `each()` and `animate` the wrapped set directly.

Within the iterator function, the `animate()` method is applied to the element (identified via `this`) with style property values for `width` and `height` set to double the element's original dimensions. The result is that over the course of two seconds (as specified by the `duration` parameter of 2000 ❸), the wrapped elements (or element) will grow from their original size to twice that size.

Now let's try something a bit more extravagant.

### 5.3.2 A custom drop animation

Let's say that we want to conspicuously animate the removal of an element from the display, perhaps because it's vitally important to convey to users that the item being removed is *gone* and that they should make no mistake about it. The animation we'll use to accomplish this will make it appear as if the element *drops* off the page, disappearing from the display as it does so.

If we think about it for a moment, we can figure out that by adjusting the `top` position of the element we can make it move down the page to simulate the drop; adjusting the `opacity` will make it seem to vanish as it does so. And finally, when all

that's done, we can remove the element from the display (similar to the animated `hide()` method).

We can accomplish this drop effect with the code in listing 5.5.

#### Listing 5.5 A custom drop animation

```

$( '.animateMe' ).each(function(){
  $(this)
    .css('position', 'relative')
    .animate(
      {
        opacity: 0,
        top: $(window).height() - $(this).height() -
            $(this).position().top
      },
      'slow',
      function(){ $(this).hide(); }
    );
});

```

1 Dislodges element from static flow

2 Computes drop distance

3 Removes element from display

There's a bit more going on here than in our previous custom effect. We once again iterate over the element set, this time adjusting the position *and* opacity of the elements. But to adjust the `top` value of an element relative to its original position, we first need to change its CSS position style property value to `relative` 1.

Then, for the animation, we specify a target opacity of 0 and a computed `top` value. We don't want to move an element so far down the page that it moves below the window's bottom; this could cause scroll bars to be displayed where none may have been before, possibly distracting users. We don't want to draw their attention away from the animation—grabbing their attention is why we're animating in the first place! So we use the height and vertical position of the element, as well as the height of the window, to compute how far down the page the element should drop 2.

When the animation is complete, we want to remove the element from the display, so we specify a callback routine 3 that applies the non-animated `hide()` method to the element (which is available to the function as its function context).

**NOTE** We did a little more work than we needed to in this animation, just so we could demonstrate doing something that needs to wait until the animation is complete in the callback function. If we were to specify the value of the opacity property as "hide" rather than 0, the removal of the element(s) at the end of the animation would be automatic, and we could dispense with the callback.

Now let's try one more type of "make it go away" effect for good measure.

### 5.3.3 A custom puff animation

Rather than dropping elements off the page, let's say we want an effect that makes it appear as if the element dissipates away into thin air like a puff of smoke. To animate such an effect, we can combine a scale effect with an opacity effect, growing the element while fading it away. One issue we need to deal with for this effect is that this

dissipation would not fool the eye if we let the element grow in place with its upper-left corner anchored. We want the *center* of the element to stay in the same place as it grows, so in addition to its size we also need to adjust the *position* of the element as part of the animation.

The code for our puff effect is shown in listing 5.6.

#### Listing 5.6 A custom puff animation

```
$('.animateMe').each(function(){
    var position = $(this).position();
    $(this)
        .css({position: 'absolute',
              top: position.top,
              left: position.left})
        .animate(
            {
                opacity: 'hide',
                width: $(this).width() * 5,
                height: $(this).height() * 5,
                top: position.top - ($(this).height() * 5 / 2),
                left: position.left - ($(this).width() * 5 / 2)
            },
            'normal');
});
```

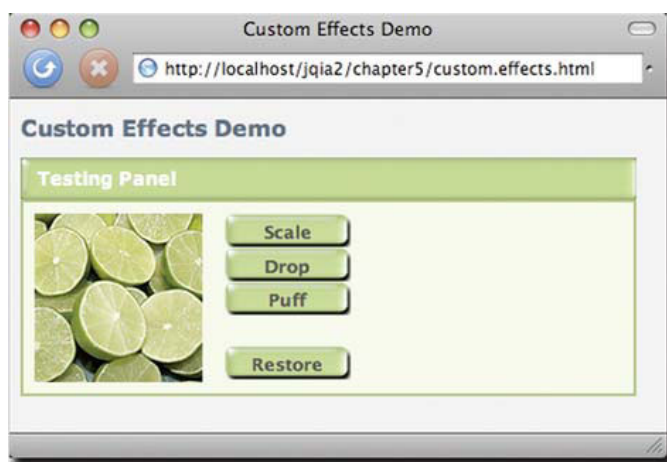
1 Dislodges element from static flow

2 Adjusts element size, position, opacity

In this animation, we decrease the opacity to 0 while growing the element to five times its original size and adjusting its position by half that new size, resulting in the center of the element remaining in the same position 2. We don't want the elements surrounding the animated element to be pushed out while the target element is growing, so we take it out of the layout flow completely by changing its position to absolute and explicitly setting its position coordinates 1.

Because we specified "hide" for the opacity value, the elements are automatically hidden (removed from the display) once the animation is complete.

Each of these three custom effects can be observed by loading the page at `chapter5/custom.effects.html`, whose display is shown in figure 5.4.



**Figure 5.4** The custom effects we developed, Scale, Drop, and Puff, can be observed in action using the buttons provided on this example page.





**Figure 5.5** The Puff effect expands and moves the image while simultaneously reducing its opacity.

We purposefully kept the browser window to a minimum size for the screenshot, but you'll want to make the window bigger when running this page to properly observe the behavior of the effects. And although we'd love to show you how these effects behave, screenshots have obvious limitations. Nevertheless, figure 5.5 shows the puff effect in progress.

We'll leave it to you to try out the various effects on this page and observe their behavior.

Up until this point, all of the examples we've examined have used a single animation method. Let's discuss how things work when we use more than one.

## 5.4 Animations and Queuing

We've seen how multiple properties of elements can be animated using a single animation method, but we haven't really examined how animations behave when we call simultaneous animation methods.

In this section we'll examine just how animations behave in concert with each other.

### 5.4.1 Simultaneous animations

What would you expect to happen if we were to execute the following code?

```
$('#testSubject').animate({left:'+=256'}, 'slow');
$('#testSubject').animate({top:'+=256'}, 'slow');
```

We know that the `animate()` method doesn't block while its animation is running on the page; nor do any of the other animation methods. We can prove that to ourselves by experimenting with this code block:



```
say(1);
$('#testSubject').animate({left:'+=256'}, 'slow');
say(2);
```

Recall that we introduced the `say()` function in chapter 4 as a way to spit messages onto an on-page “console” in order to avoid alerts (which would most definitely screw up our observation of the experiment).

If we were to execute this code, we’d see that the messages “1” and “2” are emitted immediately, one after the other, without waiting for the animation to complete.

So, what would we expect to happen when we run the code with two animation method calls? Because the second method isn’t blocked by the first, it stands to reason that both animations fire off simultaneously (or within a few milliseconds of each other), and that the effect on the test subject would be the combination of the two effects. In this case, because one effect is adjusting the `left` style property, and the other the `top` style property, we might expect that the result would be a meandering diagonal movement of the test subject.



Let’s put that to the test. In file `chapter5/revolutions.html` we’ve put together an experiment that sets up two images (one of which is to be animated), a button to start the experiment, and a “console” in which the `say()` function will write its output. Figure 5.6 shows its initial state.

The Start button is instrumented as shown in listing 5.7.

#### Listing 5.7 Instrumentation for multiple simultaneous animations

```
$('#startButton').click(function(){
    say(1);
    $('img[alt='moon']').animate({left:'+=256'}, 2500);
    say(2);
    $('img[alt='moon']').animate({top:'+=256'}, 2500);
```



**Figure 5.6** Initial state of the page where we’ll observe the behavior of multiple, simultaneous animations

```
say(3);
$("#img[alt='moon']").animate({left:'-=256'},2500);
say(4);
$("#img[alt='moon']").animate({top:'-=256'},2500);
say(5);
});
```

In the click handler for the button, we fire off four animations, one after the other, interspersed with calls to `say()` that show us when the animation calls were fired off.

Bring up the page, and click the Start button.

As expected, the console messages “1” through “5” immediately appear on the console as shown in figure 5.7, each firing off a few milliseconds after the previous one.

But what of the animations? If we examine the code in listing 5.7, we can see that we have two animations changing the `top` property and two animations changing the `left` property. In fact, the animations for each property are doing the exact opposite of each other. So what should we expect? Might they just cancel each other out, leaving the Moon (our test subject) to remain completely still?

No. Upon clicking Start, we see that each animation happens serially, one after the other, such that the Moon makes a complete and orderly revolution around the Earth (albeit in a very unnatural square orbit that would have made Kepler’s head explode).

What’s going on? We’ve proven via the console messages that the animations aren’t blocking, yet they execute serially just as if they were (at least with respect to each other).

What’s happening is that, internally, jQuery is queuing up the animations and executing them serially on our behalf.

Refresh the Kepler’s Dilemma page to clear the console, and click the Start button three times in succession. (Pause between clicks just long enough to avoid double-clicks.) You’ll note how 15 messages get immediately sent to the console, indicating that our click handler has executed three times, and then sit back as the Moon makes three orbits around the Earth.

Each of the 12 animations is queued up by jQuery and executed in order. jQuery maintains a queue on each animated element named `fx` just for this purpose. (The significance of the queue having a name will become clear in the next section.)

The queuing of animations in this manner means that we can have our cake and eat it too! We can affect multiple properties simultaneously by using a single `animate()` method that specifies all the animated properties, and we can serially execute any animations we want by simply calling them in order.

What’s even better is that jQuery makes it possible for us to create our own execution queues, not just for animations, but for whatever purposes we want. Let’s learn about that.



**Figure 5.7** The console messages appear in rapid succession, proving that the animation methods aren’t blocking until completion.

### 5.4.2 Queuing functions for execution

Queuing up animations for serial execution is an obvious use for function queues. But is there a real benefit? After all, the animation methods allow for completion callbacks, so why not just fire off the next animation in the callback of the previous animation?

#### ADDING FUNCTIONS TO A QUEUE

Let's review the code fragment of listing 5.7 (minus the `say()` invocations for clarity):

```
$("img[alt='moon']").animate({left: '+=256'}, 2500);
$("img[alt='moon']").animate({top: '+=256'}, 2500);
$("img[alt='moon']").animate({left: '-=256'}, 2500);
$("img[alt='moon']").animate({top: '-=256'}, 2500);
```

Compare that to the equivalent code that would be necessary without function queuing, using the completion callbacks:

```
$('#startButton').click(function(){
    $("img[alt='moon']").animate({left: '+=256'}, 2500, function(){
        $("img[alt='moon']").animate({top: '+=256'}, 2500, function(){
            $("img[alt='moon']").animate({left: '-=256'}, 2500, function(){
                $("img[alt='moon']").animate({top: '-=256'}, 2500);
            });
        });
    });
});
```

It's not that the callback variant of the code is that much more complicated, but it'd be hard to argue that the original code isn't a lot easier to read (and to write in the first place). And if the bodies of the callback functions were to get substantially more complicated ... Well, it's easy to see how being able to queue up the animations makes the code a lot less complex.

So what if we wanted to do the same thing with our own functions? Well, jQuery isn't greedy about its queues; we can create our own to queue up any functions we'd like to have executed in serial order.

Queues can be created on any element, and distinct queues can be created by using unique names for them (except for `fx` which is reserved for the effects queue). The method to add a function instance to a queue is, unsurprisingly, `queue()`, and it has three variants:

#### Method syntax: queue

**queue(name)**

**queue(name, function)**

**queue(name, queue)**

The first form returns any queue of the passed name already established on the first element in the matched set as an array of functions.

The second form adds the passed function to the end of the named queue for all elements in the matched set. If such a named queue doesn't exist on an element, it's created.

The last form replaces any existing queue on the matched elements with the passed queue.

**Method syntax: queue (continued)****Parameters**

name	(String) The name of the queue to be fetched, added to, or replaced. If omitted, the default effects queue of <code>fx</code> is assumed.
function	(Function) The function to be added to the end of the queue. When invoked, the function context ( <code>this</code> ) will be set to the DOM element upon which the queue has been established.
queue	(Array) An array of functions that <i>replaces</i> the existing functions in the named queue.

**Returns**

An array of functions for the first form, and the wrapped set for the remaining forms.

The `queue()` method is most often used to add functions to the end of the named queue, but it can also be used to fetch any existing functions in a queue, or to replace the list of functions in a queue. Note that the array form, in which an array of functions is passed to `queue()`, can't be used to add multiple functions to the *end* of a queue because any existing queued functions are removed. (In order to add functions to the queue, we'd fetch the array of functions, merge the new functions, and set the modified array back into the queue.)

**EXECUTING THE QUEUED FUNCTIONS**

OK, so now we can queue functions up for execution. That's not all that useful unless we can somehow cause the execution of the functions to actually occur. Enter the `dequeue()` method.

**Method syntax: dequeue****dequeue (name)**

Removes the foremost function in the named queue for each element in the matched set and executes it for each element.

**Parameters**

name	(String) The name of the queue from which the foremost function is to be removed and executed. If omitted, the default effects queue of <code>fx</code> is assumed.
------	---

**Returns**

The wrapped set.

When `dequeue()` is invoked, the foremost function in the named queue for each element in the wrapped set is executed with the function context for the invocation (`this`) being set to the element.

Let's consider the code in listing 5.8.

**Listing 5.8 Queuing and dequeuing functions on multiple elements**

```
<html>
  <head>
    <link rel="stylesheet" type="text/css" href="../../styles/core.css" />
    <script type="text/javascript" src="../../scripts/jquery-1.4.js"></script>
    <script type="text/javascript" src="console.js"></script>
    <script type="text/javascript">
```

```

$(function() {
    $('img').queue('chain',
        function(){ say('First: ' + $(this).attr('alt')); });
    $('img').queue('chain',
        function(){ say('Second: ' + $(this).attr('alt')); });
    $('img').queue('chain',
        function(){ say('Third: ' + $(this).attr('alt')); });
    $('img').queue('chain',
        function(){ say('Fourth: ' + $(this).attr('alt')); });

    $('button').click(function() {
        $('img').dequeue('chain');
    });
});
</script>
</head>

<body>

<div>
    
    
</div>

<button type="button" class="green90x24">Dequeue</button>

<div id="console"></div>

</body>
</html>

```

1  
Establishes  
four queued  
functions

2  
Dequeues one  
function on  
each click

In this example (found in file `chapter5/queue.html`), we have two images upon which we establish queues named `chain`. In each queue, we place four functions ❶ that identify themselves in order and emit the `alt` attribute of whatever DOM element is serving as the function context. This way, we can tell which function is being executed, and from which element's queue.

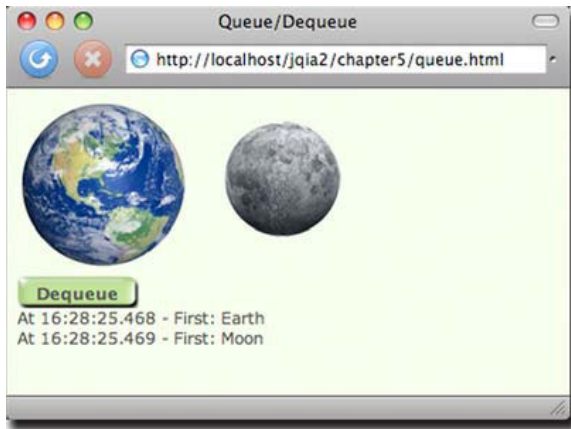
Upon clicking the Dequeue button, the button's click handler ❷ causes a single execution of the `dequeue()` method.

Go ahead and click the button once, and observe the messages in the console, as shown in figure 5.8.

We can see that the first function we added to the `chain` queue for the images has been fired twice: once for the Earth image, and once for the Moon image.

Clicking the button more times removes the subsequent functions from the queues one at a time, and executes them until the queues have been emptied; after which, calling `dequeue()` has no effect.

In this example, the dequeuing of the functions was under manual control—we needed to click the button four times (resulting in four calls to `dequeue()`) to get all four functions executed. Frequently we may want to trigger the execution of the entire set of queued functions. For such times, a commonly used idiom is to call the `dequeue()` method *within* the queued function in order to trigger the execution of (in other words, create a chain to) the next queued function.



**Figure 5.8** Clicking the Dequeue button causes a single queued instance of the function to fire, once for each image that it was established upon.

Consider the following changes to the code in listing 5.8:

```
$('.img').queue('chain',
function(){
    say('First: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
});
$('.img').queue('chain',
function(){
    say('Second: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
});
$('.img').queue('chain',
function(){
    say('Third: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
});
$('.img').queue('chain',
function(){
    say('Fourth: ' + $(this).attr('alt'));
    $(this).dequeue('chain');
});
```

We've made just such a change to the example in file `chapter5/queue.2.html`. Bring up that page in your browser, and click the Dequeue button. Note how the single click now triggers the execution of the entire chain of queued functions.

#### CLEARING OUT UNEXECUTED QUEUED FUNCTIONS

If we want to remove the queued functions from a queue without executing them, we can do that with the `clearQueue()` method:

**Method syntax: clearQueue****clearQueue(name)**

Removes all unexecuted functions from the named queue.

**Parameters**

**name** (String) The name of the queue from which the functions are to be removed without execution. If omitted, the default effects queue of `fx` is assumed.

**Returns**

The wrapped set.

While similar to the `stop()` animation method, `clearQueue()` is intended for use on general queued functions rather than just animation effects.

**DELAYING QUEUED FUNCTIONS**

Another queue-oriented activity we might want to perform is to add a delay between the execution of queued functions. The `delay()` method enables that:

**Method syntax: delay****delay(duration, name)**

Adds a delay to all unexecuted functions in the named queue.

**Parameters**

**duration** (Number|String) The delay duration in milliseconds, or one of the strings `fast` or `slow`, representing values of 200 and 600 respectively.

**name** (String) The name of the queue from which the functions are to be removed without execution. If omitted, the default effects queue of `fx` is assumed.

**Returns**

The wrapped set.

There's one more thing to discuss regarding queuing functions before moving along ...

**5.4.3 Inserting functions into the effects queue**

We previously mentioned that internally jQuery uses a queue named `fx` to queue up the functions necessary to implement the animations. What if we'd like to add our own functions to this queue in order to intersperse actions within a queued series of effects? Now that we know about the queuing methods, we can!

Think back to our previous example in listing 5.7, where we used four animations to make the Moon revolve around the Earth. Imagine that we wanted to turn the background of the Moon image black after the second animation (the one that moves it downward). If we just added a call to the `css()` method between the second and third animations, as follows,

```
$("img[alt='moon']").animate({left: '+=256'}, 2500);
$("img[alt='moon']").animate({top: '+=256'}, 2500);
$("img[alt='moon']").css({'backgroundColor': 'black'});
$("img[alt='moon']").animate({left: '-=256'}, 2500);
$("img[alt='moon']").animate({top: '-=256'}, 2500);
```



we'd be very disappointed because this would cause the background to change immediately, perhaps even before the first animation had a chance to start.

Rather, consider the following code:

```
$("img[alt='moon']").animate({left:'+=256'},2500);
$("img[alt='moon']").animate({top:'+=256'},2500);
$("img[alt='moon']").queue('fx',
function(){
    $(this).css({'backgroundColor':'black'});
    $(this).dequeue('fx');
});
$("img[alt='moon']").animate({left:'-=256'},2500);
$("img[alt='moon']").animate({top:'-=256'},2500);
```

Here, we wrap the `css()` method in a function that we place onto the `fx` queue using the `queue()` method. (We could have omitted the queue name, because `fx` is the default, but we made it explicit here for clarity.) This puts our color-changing function into place on the effects queue where it will be called as part of the function chain that executes as the animations progress, between the second and third animations.

*But note!* After we call the `css()` method, we call the `dequeue()` method on the `fx` queue. This is absolute necessary to keep the animation queue chugging along. Failure to call `dequeue()` at this point will cause the animations to grind to a halt, because nothing is causing the next function in the chain to execute. The unexecuted animations will just sit there on the effects queues until either something causes a `dequeue` and the functions commence, or the page unloads and everything just gets discarded.

If you'd like to see this process in action, load the page in file `chapter5/revolutions.2.html` into your browser and click the button.

Queuing functions comes in handy whenever we want to execute functions consecutively, but without the overhead, or complexity, of nesting functions in asynchronous callbacks; something that, as you might imagine, can come in handy when we throw Ajax into the equation.

But that's another chapter.

## 5.5 Summary

This chapter introduced us to the animated effects that jQuery makes available out of the box, as well as to the `animate()` method that allows us to create our own custom animations.

The `show()` and `hide()` methods, when used without parameters, reveal and conceal elements from the display immediately, without any animation. We can perform animated versions of the hiding and showing of elements with these methods by passing parameters that control the speed of the animation, as well as providing an optional callback that's invoked when the animation completes. The `toggle()` method toggles the displayed state of an element between hidden and shown.

Another set of wrapper methods, `fadeOut()` and `fadeIn()`, also hides and shows elements by adjusting the opacity of elements when removing or revealing them in



the display. A third method, `fadeOut()`, animates a change in opacity for its wrapped elements without removing the elements from the display.

A final set of three built-in effects animates the removal or display of our wrapped elements by adjusting their vertical height: `slideUp()`, `slideDown()`, and `slideToggle()`.

For building our own custom animations, jQuery provides the `animate()` method. Using this method, we can animate any CSS style property that accepts a numeric value, most commonly the opacity, position, and dimensions of the elements. We explored writing some custom animations that remove elements from the page in novel fashions.

We also learned how jQuery queues animations for serial execution, and how we can use the jQuery queuing methods to add our own functions to the effects queue or our own custom queues.

When we explored writing our own custom animations, we wrote the code for these custom effects as inline code within the on-page JavaScript. A much more common, and useful, method is to package custom animations as custom jQuery methods. We'll learn how to do that in chapter 7, and you're encouraged to revisit these effects after you've read that chapter. Repackaging the custom effects we developed in this chapter, and any that you can think up on your own, would be an excellent follow-up exercise.

But before we write our own jQuery extensions, let's take a look at some high-level jQuery functions and flags that will be very useful for general tasks as well as extension writing.