4
Styling and Animating

If actions speak louder than words, then in the JavaScript world, effects make actions speak louder still. With jQuery, we can easily add impact to our actions through a set of simple visual **effects**, and even craft our own, more sophisticated **animations**.

The effects offered by jQuery supply simple visual flourishes that grant a sense of movement and modernity to any page. However, apart from being mere decoration, they can also provide important usability enhancements that help orient the user when there is some change on a page (especially common in Ajax applications). In this chapter, we will explore a number of these effects and combine them in interesting ways.

Inline CSS modification

Before we jump into the nifty jQuery effects, a quick look at CSS is in order. In earlier chapters, we have been modifying a document's appearance by defining styles for classes in a separate stylesheet and then adding or removing those classes with jQuery. Typically, this is the preferred process for injecting CSS into HTML because it respects the stylesheet's role in dealing with the presentation of a page. However, there may be times when we need to apply styles that haven't been, or can't easily be, defined in a stylesheet. Fortunately, jQuery offers the `.css()` method for such occasions.

This method acts as both a **getter** and a **setter**. To get the value of a style property, we simply pass the name of the property as a string, like `.css('backgroundColor')`. Multi-word properties such as this one can be interpreted by jQuery when in hyphenated CSS notation (`background-color`), or camel-cased DOM notation (`backgroundColor`). For setting-style properties, the `.css()` method comes in two flavors—one that takes a single style property and its value, and one that takes a **map** of property-value pairs, as shown in the following code snippet:

```
// Single property and its value
.css('property', 'value')
```

```
// Map of property-value pairs
.css({
  property1: 'value1',
  'property-2': 'value2'
})
```

Experienced JavaScript developers will recognize these jQuery maps as JavaScript **object literals**.

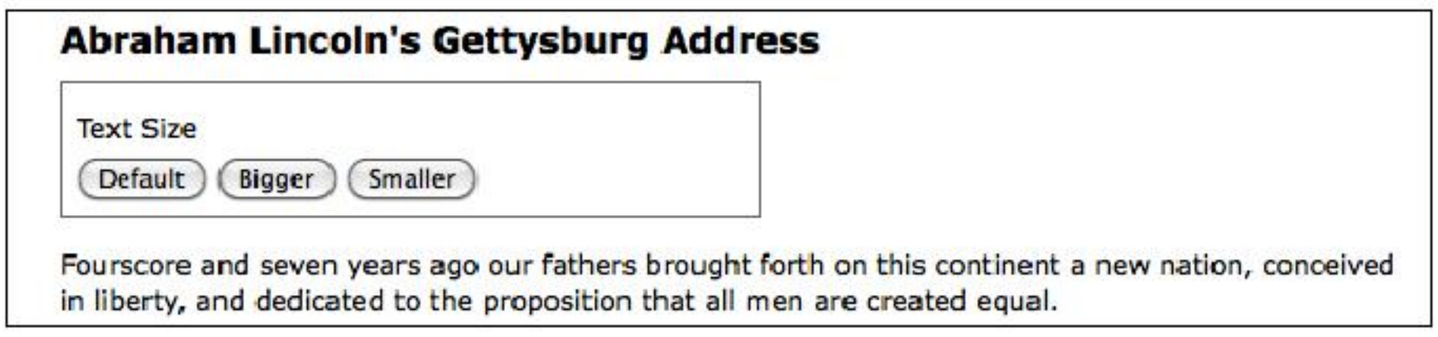
Object literal notation

In a property value, strings are enclosed in quotes as usual, but other data types such as numbers do not require them. Since property names are strings, they typically would be contained in quotes. However, quotation marks are not required for property names if they are valid JavaScript identifiers, such as when they are written in camel-cased DOM notation.

We use the `.css()` method the same way we've been using `.addClass()`: we apply it to a jQuery object, which in turn points to a collection of DOM elements. To demonstrate this, we'll play with a style switcher similar to the one from *Chapter 3, Handling Events*:

```
<div id="switcher">
  <div class="label">Text Size</div>
  <button id="switcher-default">Default</button>
  <button id="switcher-large">Bigger</button>
  <button id="switcher-small">Smaller</button>
</div>
<div class="speech">
  <p>Fourscore and seven years ago our fathers brought forth
    on this continent a new nation, conceived in liberty,
    and dedicated to the proposition that all men are created
    equal.
  </p>
</div>
```


By linking to a stylesheet with a few basic style rules, the page will initially look like the following screenshot:



Once we're done with our code, clicking on the **Bigger** and **Smaller** buttons will increase or decrease , respectively, the text size of `<div class="speech">`, while clicking on the **Default** button will reset `<div class="speech">` to its original text size.

If all we wanted were to change the font size a single time to a predetermined value, then we could still use the `.addClass()` method. However, let's suppose that now we want the text to continue increasing or decreasing incrementally each time the respective button is clicked. Although it might be possible to define a separate class for each click and iterate through them, a more straightforward approach would be to compute the new text size each time by getting the current size and increasing it by a set factor (for example, 40%).

Our code will start with the `$(document).ready()` and `$('#switcher-large').click()` event handlers, as follows:

```
$(document).ready(function() {
  $('#switcher-large').click(function() {
  });
});
```

Listing 4.1

Next, the font size can be easily discovered by using the `.css()` method: `$('div.speech').css('fontSize')`. However, the returned value is a string, containing both the numeric font size value and the units of that value (`px`). We'll need to strip the unit label off in order to perform calculations with the numeric value. Also, when we plan to use a jQuery object more than once, it's generally a good idea to **cache** the selector by storing the resulting jQuery object in a variable. We'll take care of these needs with the introduction of a couple of local variables:


```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher-large').click(function() {
    var num = parseFloat($speech.css('fontSize'));
  });
});
```

Listing 4.2

The first line inside `$(document).ready()` now creates a variable containing a jQuery object pointing to `<div class="speech">`. Notice the use of a `$` in the variable name, `$speech`. As `$` is a legal character in JavaScript identifiers, we can use it as a reminder that the variable is storing a jQuery object.

Inside the `.click()` handler, we use `parseFloat()` to get the font size property's numeric value only. The `parseFloat()` function looks at a string from left to right until it encounters a non-numeric character. The string of digits is converted into a floating-point (decimal) number. For example, it would convert the string `'12'` to the number `12`. In addition, it strips non-numeric trailing characters from the string, so `'12px'` becomes `12` as well. If the string begins with a non-numeric character, `parseFloat()` returns `NaN`, which stands for **Not a Number**.

All that's left to do is to modify the parsed numeric value and to reset the font size based on the new value. For our example, we'll increase the font size by 40% each time the button is clicked. To achieve this, we'll multiply `num` by `1.4` and then set the font size by concatenating `num` and `'px'`, as shown in the following code snippet:

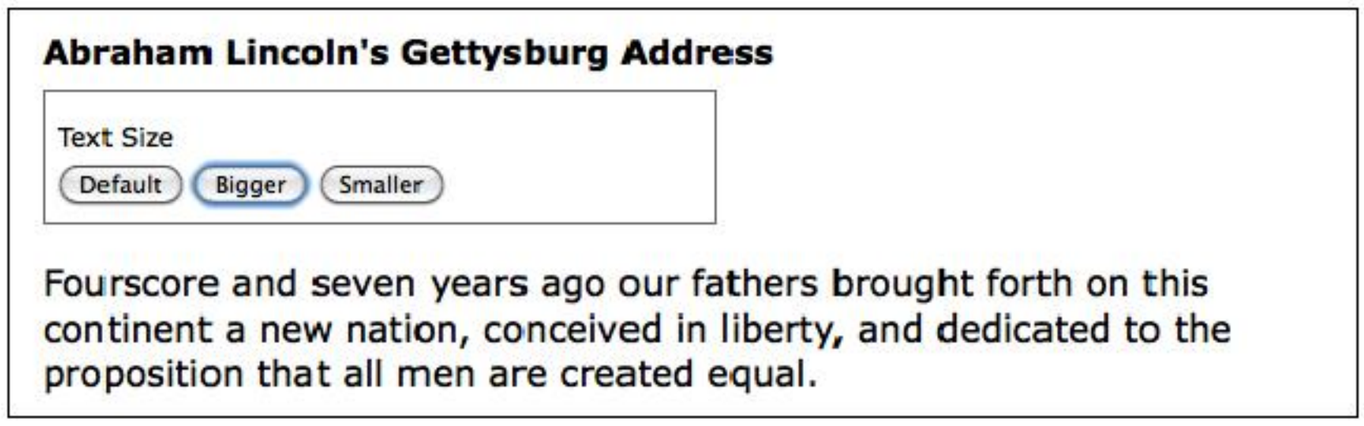

```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher-large').click(function() {
    var num = parseFloat($speech.css('fontSize'));
    num *= 1.4;
    $speech.css('fontSize', num + 'px');
  });
});
```

Listing 4.3

Shorthand operators

The equation `num *= 1.4` is shorthand for `num = num * 1.4`. We can use the same type of shorthand for the other basic mathematical operations, as well: addition (`a += b`), subtraction (`a -= b`), division (`a /= b`), and modulus/remainder (`a %= b`).

Now when a user clicks on the **Bigger** button, the text becomes larger. Another click and the text becomes even larger, as shown in the following screenshot:



To get the **Smaller** button to decrease the font size, we will divide rather than multiply—`num /= 1.4`. Better still, we'll combine the two into a single `.click()` handler on all `<button>` elements within `<div id="switcher">`. Then, after finding the numeric value, we can either multiply or divide depending on the ID of the button that was clicked. The following code snippet, Listing 4.4, illustrates this:

```
$(document).ready(function() {
  var $speech = $('div.speech');
  $('#switcher button').click(function() {
    var num = parseFloat($speech.css('fontSize'));
    if (this.id == 'switcher-large') {
      num *= 1.4;
    } else if (this.id == 'switcher-small') {
      num /= 1.4;
    }
    $speech.css('fontSize', num + 'px');
  });
});
```

Listing 4.4

Recall from *Chapter 3* that we can access the `id` property of the DOM element referred to by `this`, which appears here inside the `if` and `else if` statements. Here, it is more efficient to use `this` than to create a jQuery object just to test the value of a property.

It would also be nice to have a way to return the font size to its initial value. To allow the user to do so, we can simply store the font size in a variable immediately when the DOM is ready. We can then restore this value whenever the **Default** button is clicked. To handle this click, we could add another `else if` statement. A `switch` statement may be more appropriate, as shown in the following code snippet:

```
$(document).ready(function() {
  var $speech = $('div.speech');
  var defaultSize = $speech.css('fontSize');
  $('#switcher button').click(function() {
    var num = parseFloat($speech.css('fontSize'));
    switch (this.id) {
      case 'switcher-large':
        num *= 1.4;
        break;
      case 'switcher-small':
        num /= 1.4;
        break;
      default:
```

```
            num = parseFloat(defaultSize);
        }
        $speech.css('fontSize', num + 'px');
    });
});
```

Listing 4.5

Here we're still checking the value of `this.id` and changing the font size based on it, but if its value is neither `'switcher-large'` nor `'switcher-small'` it will default to the initial font size.

Basic hide and show

The basic `.hide()` and `.show()` methods, without any parameters, can be thought of as smart shorthand methods for `.css('display', 'string')`, where `'string'` is the appropriate display value. The effect, as might be expected, is that the matched set of elements will be immediately hidden or shown, with no animation.

The `.hide()` method sets the **inline style attribute** of the matched set of elements to `display: none`. The smart part here is that it remembers the value of the `display` property—typically `block` or `inline`—before it was changed to `none`. Conversely, the `.show()` method restores display properties of the matched set of elements to whatever they initially were before `display: none` was applied.

The display property

For more information about the `display` property and how its values are visually represented in a web page, visit the Mozilla Developer Center at `https://developer.mozilla.org/en/CSS/display/` and view examples at `https://developer.mozilla.org/samples/cssref/display.html`.

This feature of `.show()` and `.hide()` is especially helpful when hiding elements whose default `display` property is overridden in a stylesheet. For example, the `<li>` element has the property `display: block` by default, but we might want to change it to `display: inline` for a horizontal menu. Fortunately, using the `.show()` method on a hidden element such as one of these `<li>` tags would not merely reset it to its default `display: block`, because that would put the `<li>` on its own line. Instead, the element is restored to its previous `display: inline` state, thus preserving the horizontal design.

We can set up a quick demonstration of these two methods by working with a second paragraph and a **read more** link after the first paragraph in the example HTML, as follows:

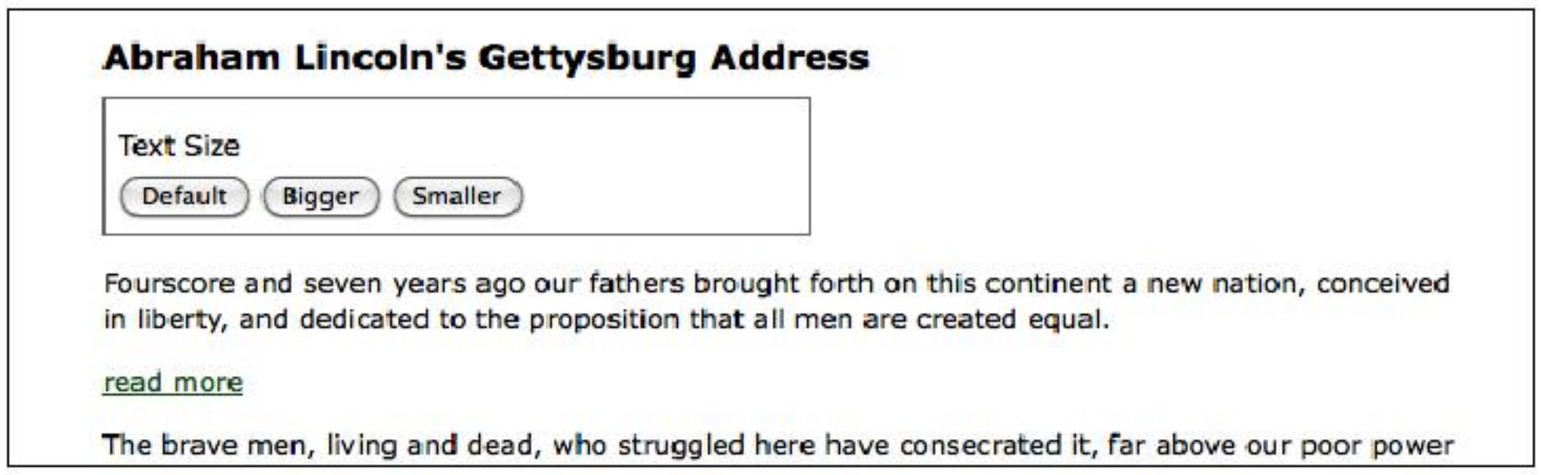
```
<div class="speech">
  <p>Fourscore and seven years ago our fathers brought forth
    on this continent a new nation, conceived in liberty,
    and dedicated to the proposition that all men are
    created equal.
  </p>
  <p>Now we are engaged in a great civil war, testing whether
    that nation, or any nation so conceived and so dedicated,
    can long endure. We are met on a great battlefield of
    that war. We have come to dedicate a portion of that
    field as a final resting-place for those who here gave
    their lives that the nation might live. It is altogether
    fitting and proper that we should do this. But, in a
    larger sense, we cannot dedicate, we cannot consecrate,
    we cannot hallow, this ground.
  </p>
  <a href="#" class="more">read more</a>
</div>
```

When the DOM is ready, we select an element and call `.hide()` on it:

```
$(document).ready(function() {
  $('p').eq(1).hide();
});
```

Listing 4.6

The `.eq()` method is similar to the `:eq()` pseudo-class discussed in *Chapter 2, Selecting Elements*. It returns a jQuery object pointing to a single element at the provided zero-based index. In this case, the method selects the second paragraph and hides it, resulting in the following:

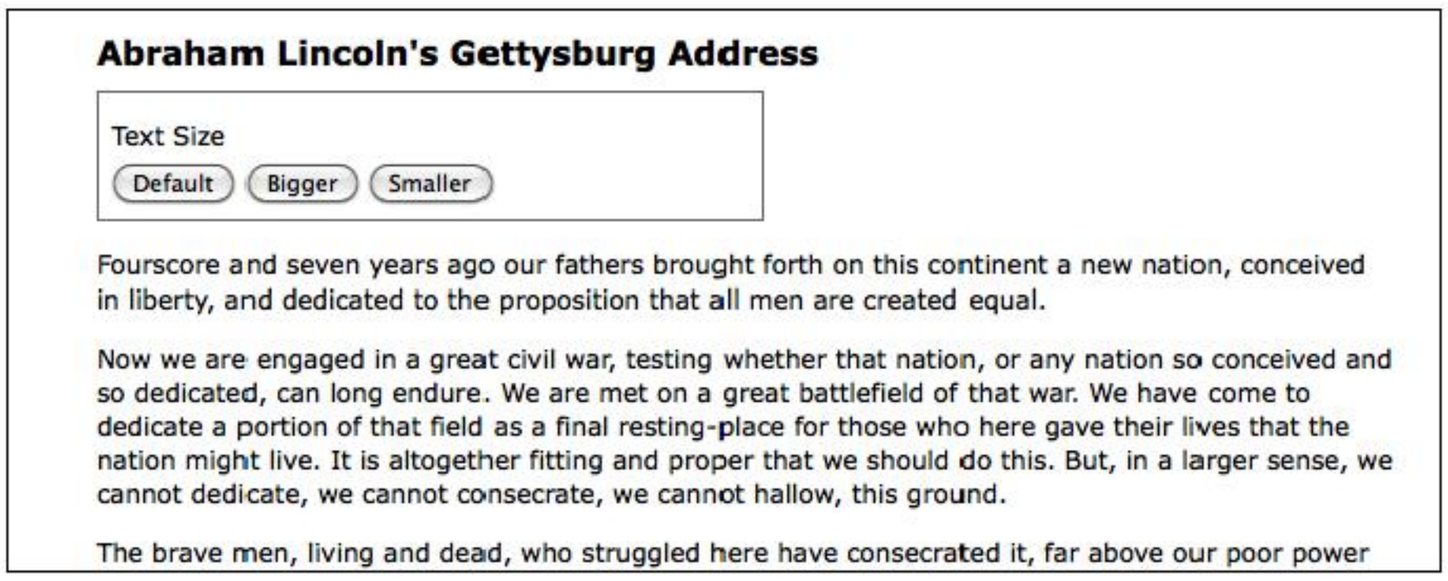


Then, when the user clicks on **read more** at the end of the first paragraph, that link is hidden and the second paragraph is shown:

```
$(document).ready(function() {
  $('p').eq(1).hide();
  $('a.more').click(function() {
    $('p').eq(1).show();
    $(this).hide();
    return false;
  });
});
```

Listing 4.7

Note the use of `return false` to keep the link from activating its default action. Now the speech looks similar to the following screenshot:



The `.hide()` and `.show()` methods are quick and useful, but they aren't very flashy. To add some flair, we can give them a speed, which is explained in the following section.

Effects and speed

When we include a **speed** (or, more precisely, a **duration**) with `.show()` or `.hide()`, it becomes animated—occurring over a specified period of time. The `.hide('speed')` method, for example, decreases an element's height, width, and opacity simultaneously until all three reach zero, at which point the CSS rule `display: none` is applied. The `.show('speed')` method will increase the element's height from top to bottom, width from left to right, and opacity from 0 to 1 until its contents are completely visible.

Speeding in

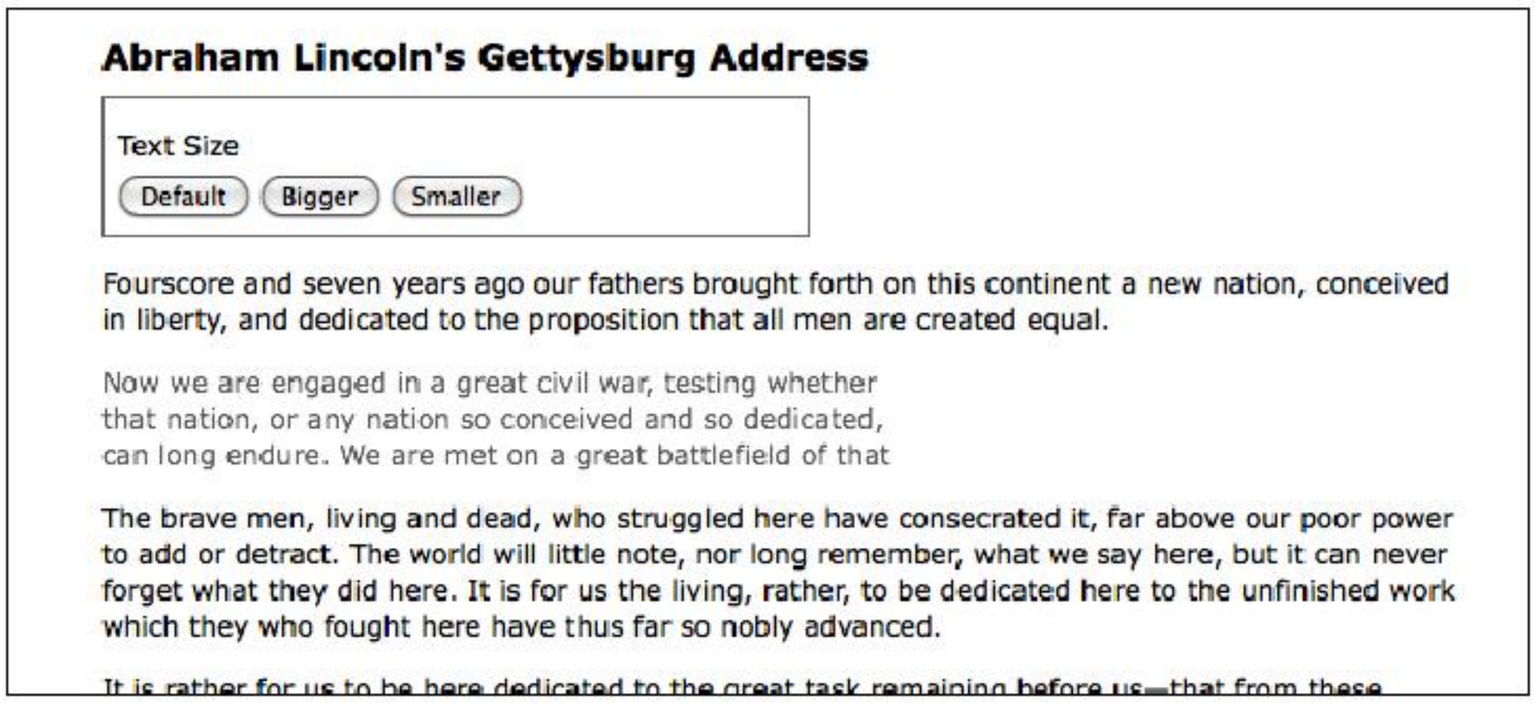
With any jQuery effect, we can use one of three preset speeds: `'slow'`, `'normal'`, and `'fast'`. Using `.show('slow')` makes the show effect complete in .6 seconds, `.show('normal')` in .4 seconds, and `.show('fast')` in .2 seconds. For even greater precision, we can specify a number of milliseconds, for example `.show(850)`. Note that in this case we are specifying a numeric value, so we do not use quotation marks.

Let's include a speed in our example when showing the second paragraph of Lincoln's Gettysburg Address:

```
$(document).ready(function() {
  $('p').eq(1).hide();
  $('a.more').click(function() {
    $('p').eq(1).show('slow');
    $(this).hide();
    return false;
  });
});
```

Listing 4.8

When we capture the paragraph's appearance at roughly halfway through the effect, we see something similar to the following screenshot:



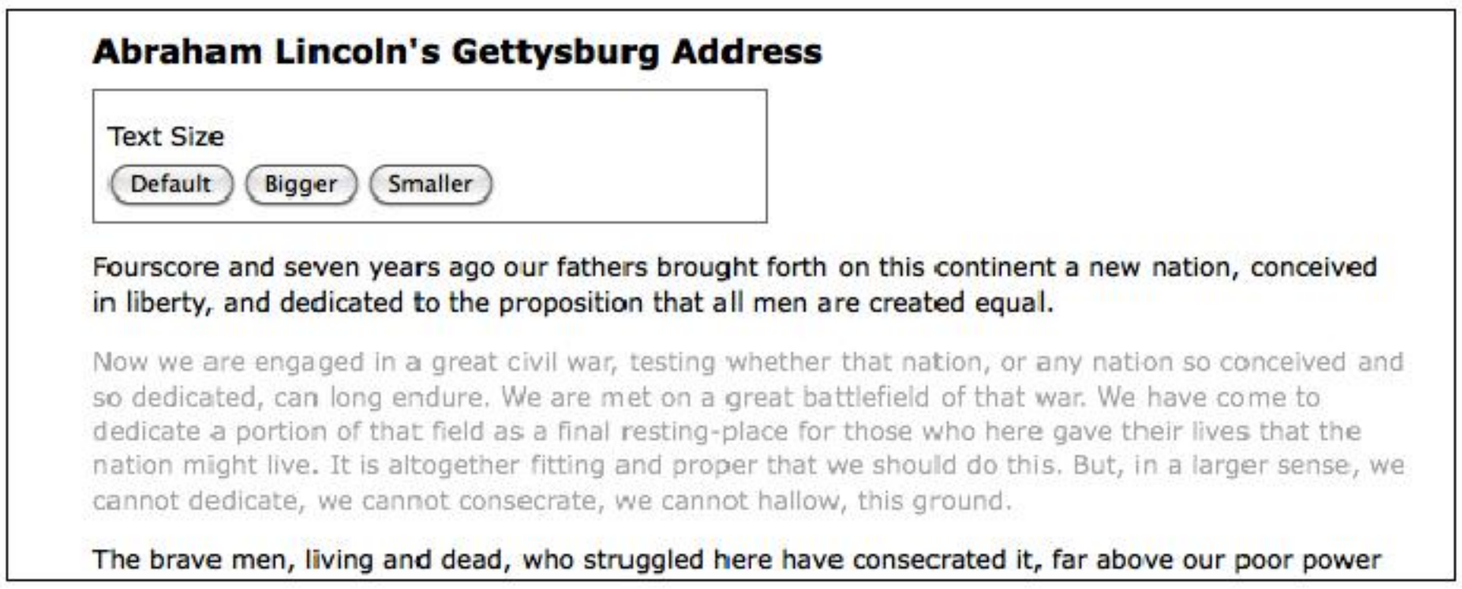
Fading in and fading out

While the animated `.show()` and `.hide()` methods are certainly flashy, in practice they animate more properties than are useful. Fortunately, jQuery offers a couple of other pre-built animations for a more subtle effect. For example, to have the whole paragraph appear just by gradually increasing the opacity, we can use `.fadeIn('slow')` instead:

```
$(document).ready(function() {
  $('p').eq(1).hide();
  $('a.more').click(function() {
    $('p').eq(1).fadeIn('slow');
    $(this).hide();
    return false;
  });
});
```

Listing 4.9

Now when we look at the paragraph during the effect, it looks similar to the following screenshot:



The difference here is that the `.fadeIn()` effect starts by setting the dimensions of the paragraph so that the contents can simply fade into it. To gradually decrease the opacity we can use `.fadeOut()`.

Sliding up and sliding down

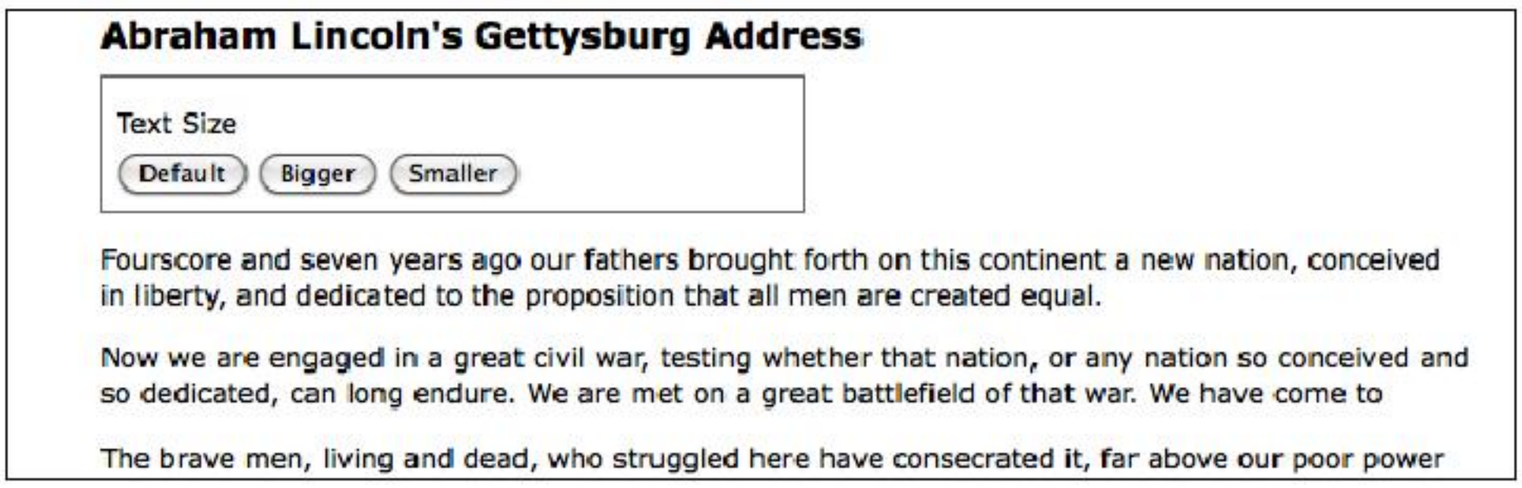
The fading animations are very useful for items that are outside the flow of the document. For example, these are typical effects to apply to "lightbox" elements that are overlaid on the page. However, when an element is part of the document flow, calling `.fadeIn()` on it causes the document to "jump" to provide the real estate needed for the new element, which is not very aesthetically pleasing.

In these cases, jQuery's `.slideDown()` and `.slideUp()` methods are often the right choice. These effects animate only the height of the selected elements. To have our paragraph appear using a vertical slide effect, we can call `.slideDown('slow')`, as follows:

```
$(document).ready(function() {
  $('p').eq(1).hide();
  $('a.more').click(function() {
    $('p').eq(1).slideDown('slow');
    $(this).hide();
    return false;
  });
});
```

Listing 4.10

This time when we examine the paragraph at the animation's midpoint, it looks similar to the following screenshot:



To reverse the effect, we would instead call `.slideUp()`.

Compound effects

Sometimes, we have a need to toggle the visibility of elements, rather than displaying them once as we have done in the preceding examples. This toggling can be achieved by first checking the visibility of the matched elements and then calling the appropriate method. Using the fade effects again, we can modify the example script to look similar to the following code snippet:

```
$(document).ready(function() {
  var $firstPara = $('p').eq(1);
  $firstPara.hide();
  $('a.more').click(function() {
    if ($firstPara.is(':hidden')) {
```

```
        $firstPara.fadeIn('slow');
        $(this).text('read less');
      } else {
        $firstPara.fadeOut('slow');
        $(this).text('read more');
      }
      return false;
    });
});
```

Listing 4.11

As we did earlier in the chapter, we're caching our selector here to avoid repeated DOM traversal. Notice, too, that we're no longer hiding the clicked link; instead, we're changing its text.

To examine the text contained by an element and to change that text, we're using the `.text()` method. This method will be more fully explored in *Chapter 5, Manipulating the DOM.*

Using an `if/else` statement is a perfectly reasonable way to toggle elements' visibility. However, with jQuery's **compound effects** we can remove some conditional logic from our code. jQuery provides a `.toggle()` method, which acts like `.show()` and `.hide()`, and like them, can be used with a speed argument or without. Other compound methods include `.fadeToggle()` and `.slideToggle()`, which show or hide elements using the corresponding effects. The following code snippet is what the script looks like when we use the `.slideToggle()` method:

```
$(document).ready(function() {
  var $firstPara = $('p').eq(1);
  $firstPara.hide();
  $('a.more').click(function() {
    $firstPara.slideToggle('slow');
    var $link = $(this);
    if ($link.text() == 'read more') {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

Listing 4.12

To reduce repetition of `$(this)`, we're storing the result in the `$link` variable for performance and readability. Also, the conditional statement checks for the text of the link rather than the visibility of the second paragraph, as we're only using it to change the text.

Creating custom animations

In addition to the pre-built effect methods, jQuery provides a powerful `.animate()` method that allows us to create our own custom animations with fine-grained control. The `.animate()` method comes in two forms. The first takes up to four arguments, which are as follows:

1. A **map** of style properties and values – similar to the `.css()` map discussed earlier in this chapter
2. An optional **speed** – which can be one of the preset strings or a number of milliseconds
3. An optional **easing type** – an advanced option discussed in *Chapter 11, Advanced Effects*
4. An optional **callback function** – which will be discussed later in this chapter

All together, the four arguments look similar to the following code snippet:

```
.animate({property1: 'value1', property2: 'value2'},
  speed, easing, function() {
    alert('The animation is finished.');
  }
);
```

The second form takes two arguments: a map of `properties` and a map of `options`:

```
.animate({properties}, {options})
```


In effect, the second argument wraps up the second through fourth arguments of the first form into another map, and adds some more advanced options to the mix. When we adjust the line breaks for readability, the second form looks similar to the following code snippet:

```
.animate({
  property1: 'value1',
  property2: 'value2'
}, {
  duration: 'value',
  easing: 'value',
  specialEasing: {
```

```
    property1: 'easing1',
    property2: 'easing2'
  },
  complete: function() {
    alert('The animation is finished.');
  },
  queue: true,
  step: callback
});
```

For now, we'll use the first form of the `.animate()` method, but we'll return to the second form later in the chapter when we discuss queuing effects.

Building effects by hand

We have already seen several pre-packaged effects for showing and hiding elements. To begin our discussion of the `.animate()` method, it will be useful to see how we could achieve the same results as calling `.slideToggle()` using this lower-level interface. Replacing the `.slideToggle()` line of the preceding example with our custom animation turns out to be quite simple, as shown in the following code snippet:

```
$(document).ready(function() {
  var $firstPara = $('p').eq(1);
  $firstPara.hide();
  $('a.more').click(function() {
    $firstPara.animate({height: 'toggle'}, 'slow');
    var $link = $(this);
    if ($link.text() == 'read more') {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

Listing 4.13

This is not a perfect replacement for `.slideToggle()`; the actual implementation also animates the margin and padding of elements.

As the example illustrates, the `.animate()` method provides convenient shorthand values for CSS properties — `'show'`, `'hide'`, and `'toggle'` — to ease the way when we want to emulate the behavior of pre-packaged effect methods such as `.slideToggle()`.

Animating multiple properties at once

With the `.animate()` method, we can modify any combination of properties simultaneously. For example, to create a simultaneous sliding and fading effect when toggling the second paragraph, we simply add the `opacity` property-value pair to `.animate()`'s properties map, as follows:


```
$(document).ready(function() {
  var $firstPara = $('p').eq(1);
  $firstPara.hide();
  $('a.more').click(function() {
    $firstPara.animate({
      opacity: 'toggle',
      height: 'toggle'
    }, 'slow');
    var $link = $(this);
    if ($link.text() == 'read more') {
      $link.text('read less');
    } else {
      $link.text('read more');
    }
    return false;
  });
});
```

Listing 4.14

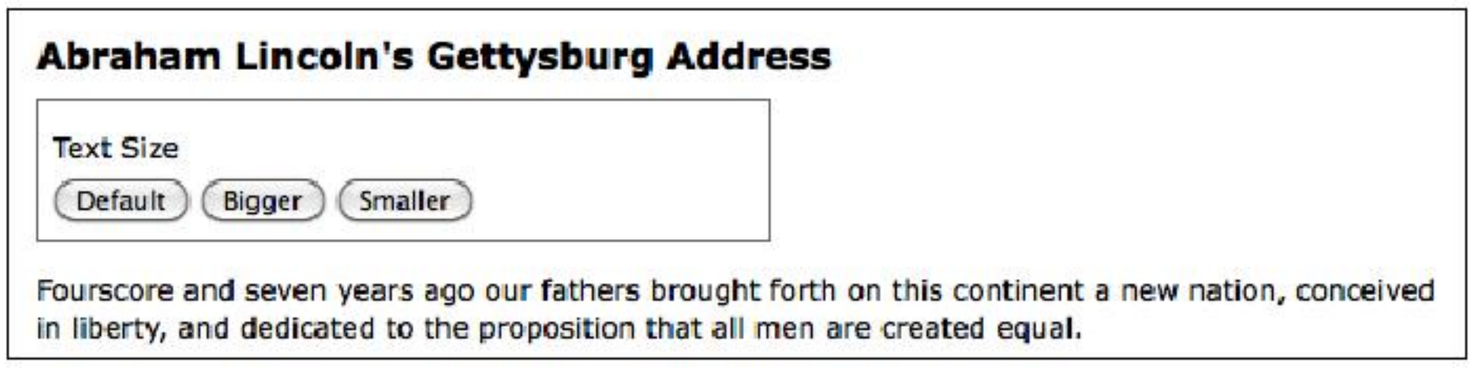
Additionally, we have not only the style properties used for the shorthand effect methods at our disposal, but numeric CSS properties, such as `left`, `top`, `fontSize`, `margin`, `padding`, and `borderWidth`. Recall Listing 4.5, which changed the text size of the speech paragraphs. We could animate the increase or decrease in size by simply substituting the `.animate()` method for the `.css()` method, as shown in the following code snippet:

```
$(document).ready(function() {
  var $speech = $('div.speech');
  var defaultSize = $speech.css('fontSize');
  $('#switcher button').click(function() {
    var num = parseFloat($speech.css('fontSize'));
```

```
        switch (this.id) {
          case 'switcher-large':
            num *= 1.4;
            break;
          case 'switcher-small':
            num /= 1.4;
            break;
          default:
            num = parseFloat(defaultSize);
        }
        $speech.animate({fontSize: num + 'px'}, 'slow');
      });
    });
```


Listing 4.15

The extra properties allow us to create much more complex effects, too. We can, for example, move an item from the left side of the page to the right while increasing its height by 20 pixels and changing its border width to 5 pixels. We will illustrate this complicated set of property animations with the `<div id="switcher">` box. The following screenshot is what it looks like before we animate it:



With a flexible-width layout, we need to compute the distance that the box needs to travel before it lines up at the right side of the page. Assuming that the paragraph's width is 100%, we can subtract the **Text Size** box's width from the paragraph's width. We have jQuery's `.outerWidth()` method at our disposal to calculate these widths, including padding and border. We'll use this method to compute the new `left` property of the switcher. For the sake of this example, we'll trigger the animation by clicking on the **Text Size** label, just above the buttons. The following code snippet is what the code should look like:

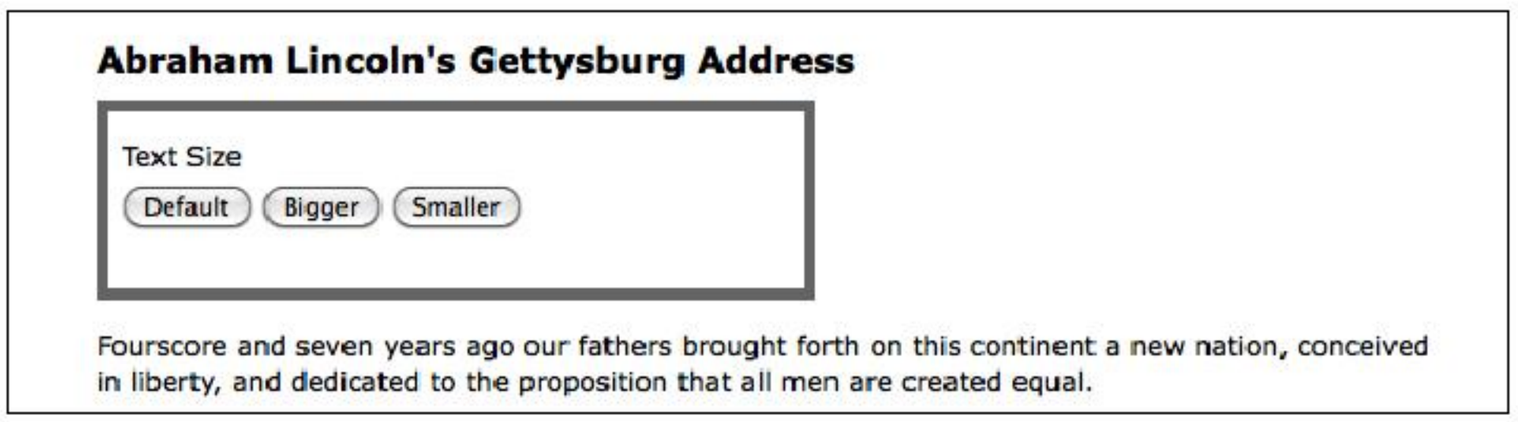
```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
```

```
      $switcher.animate({
        borderWidth: '5px',
        left: paraWidth - switcherWidth,
        height: '+=20px'
      }, 'slow');
    });
});
```

Listing 4.16

It is worth examining these animated properties in detail. The `borderWidth` property is straightforward, as we are specifying a constant value with units, just as we would in a stylesheet. The `left` property is a computed numeric value. The unit suffix is optional on these properties; as we omit it here, `px` is assumed. Finally, the `height` property uses a syntax we have not seen before. The `+=` prefix on a property value indicates a relative value. So, instead of animating the height to 20 pixels, the height is animated to 20 pixels greater than the current height. Because of the special characters involved, relative values must be specified as a string, so must be enclosed in quotes.

Although this code successfully increases the height of the `<div>` and widens its border, at the moment the `left` position remains unchanged, as shown in the following screenshot:



We still need to enable changing this box's position in the CSS.

Positioning with CSS

When working with `.animate()`, it's important to keep in mind the limitations that CSS imposes on the elements that we wish to change. For example, adjusting the `left` property will have no effect on the matching elements unless those elements have their CSS position set to `relative` or `absolute`. The default CSS position for all block-level elements is `static`, which accurately describes how those elements will remain if we try to move them without first changing their `position` value.

For more information on absolute and relative positioning, see Joe Gillespie's article, **Absolutely Relative**, at:
`http://www.wpdfd.com/issues/78/absolutely_relative/`

In our stylesheet, we could set `<div id="switcher">` to be relatively positioned:

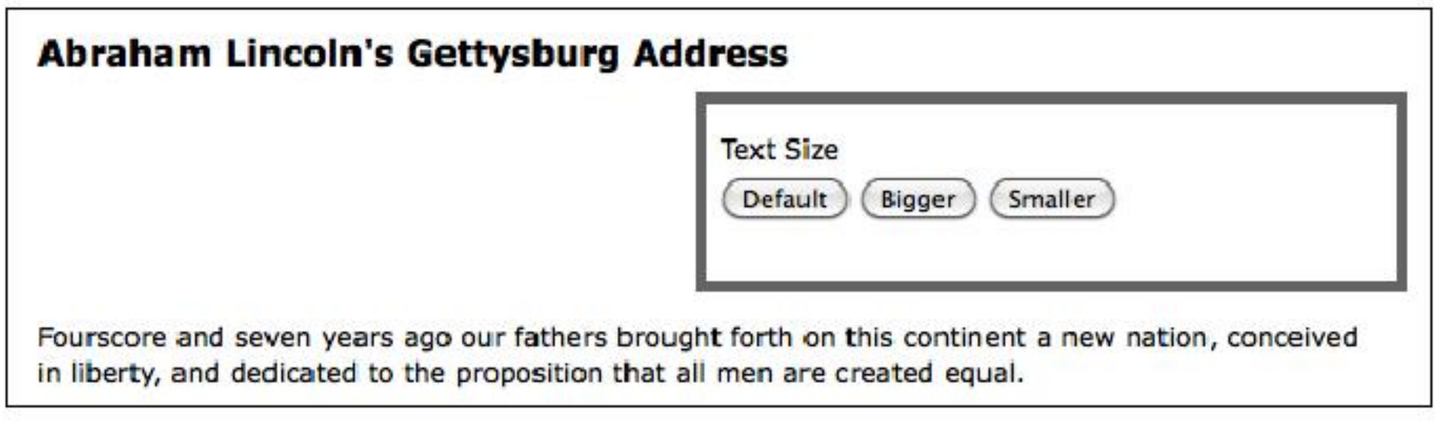
```
#switcher {
  position: relative;
}
```

Instead, though, let's practice our jQuery skills by altering this property through JavaScript when needed, as shown in the following code snippet:


```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher.css({
      position: 'relative'
    }).animate({
      borderWidth: '5px',
      left: paraWidth - switcherWidth,
      height: '+=20px'
    }, 'slow');
  });
});
```

Listing 4.17

With the CSS taken into account, the result of clicking on **Text Size**, when the animation has completed, will look similar to the following screenshot:



Simultaneous versus queued effects

The `.animate()` method, as we've just discovered, is very useful for creating simultaneous effects affecting a particular set of elements. There may be times, however, when we want to queue our effects, having them occur one after the other.

Working with a single set of elements

When applying multiple effects to the same set of elements, **queuing** is easily achieved by chaining those effects. To demonstrate this queuing, we'll repeat Listing 4.17, by moving the **Text Size** box to the right, increasing its height, and increasing its border width. This time, however, we perform the three effects sequentially, simply by placing each in its own `.animate()` method and chaining the three together:


```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .animate({left: paraWidth - switcherWidth}, 'slow')
      .animate({height: '+=20px'}, 'slow')
      .animate({borderWidth: '5px'}, 'slow');
  });
});
```

Listing 4.18

Recall that chaining permits us to keep all three `.animate()` methods on the same line, but here we have indented them and put each on its own line for greater readability.

We can queue any of the jQuery effect methods, not just `.animate()`, by chaining them. We could, for example, queue effects on `<div id="switcher">` in the following order:

1. Fade its opacity to .5 with `.fadeTo()`
2. Move it to the right with `.animate()`
3. Fade it back in to full opacity with `.fadeTo()`
4. Hide it with `.slideUp()`
5. Show it once more with `.slideDown()`

All we need to do is chain the effects in the same order in our code, as shown in the following code snippet:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .fadeTo('fast', 0.5)
      .animate({left: paraWidth - switcherWidth}, 'slow')
      .fadeTo('slow', 1.0)
      .slideUp('slow')
      .slideDown('slow');
  });
});
```

Listing 4.19

Bypassing the queue

However, what if we want to move the `<div>` to the right at the same time as it fades to half opacity? If the two animations were occurring at the same speed, then we could simply combine them into a single `.animate()` method. However, in this example, the fade is using the `'fast'` speed, while the move to the right is using the `'slow'` speed. The following code snippet is where the second form of the `.animate()` method comes in handy:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .fadeTo('fast', 0.5)
      .animate({
        left: paraWidth - switcherWidth
      }, {
        duration: 'slow',
        queue: false
      })
      .fadeTo('slow', 1.0)
```

```
        .slideUp('slow')
        .slideDown('slow');
    });
});
```

Listing 4.20

The second argument, an options map, provides the `queue` option, which when set to `false` makes the animation start simultaneously with the previous one.

Manual queueing

One final observation about queuing effects on a single set of elements is that queuing does not automatically apply to other, non-effect methods such as `.css()`. So let's suppose we wanted to change the background color of `<div id="switcher">` to red after the `.slideUp()` but before the `slideDown()`.

To do this we can use the following code snippet:

```
// Unfinished code
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .fadeTo('fast', 0.5)
      .animate({
        left: paraWidth - switcherWidth
      }, {
        duration: 'slow',
        queue: false
      })
      .fadeTo('slow', 1.0)
      .slideUp('slow')
      .css({backgroundColor: '#f00'})
      .slideDown('slow');
  });
});
```

Listing 4.21

However, even though the background-changing code is placed at the correct position in the chain, it occurs immediately upon the click.

One way we can add non-effect methods to the queue is to use the appropriately-named `.queue()` method, as follows:

```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .fadeTo('fast', 0.5)
      .animate({
        left: paraWidth - switcherWidth
      }, {
        duration: 'slow',
        queue: false
      })
      .fadeTo('slow', 1.0)
      .slideUp('slow')
      .queue(function(next) {
        $switcher.css({backgroundColor: '#f00'});
        next();
      })
      .slideDown('slow');
  });
});
```

Listing 4.22

When given a callback function, as it is here, the `.queue()` method adds the function to the queue of effects to perform on the matched elements. Within the function, we set the background color to red and then call `next()`, a function that is passed as a parameter to our callback. Including this `next()` function call allows the animation queue to pick up where it left off and complete the chain with the following `.slideDown('slow')` line. If we hadn't called `next()`, the animation would have stopped.

More information and examples for `.queue()` are available at `http://api.jquery.com/category/effects/`.

We'll discover another way to queue non-effect methods as we examine effects with multiple sets of elements.

Working with multiple sets of elements

Unlike with a single set of elements, when we apply effects to different sets, they occur at virtually the same time. To see these simultaneous effects in action, we'll slide one paragraph down while sliding another paragraph up. We'll be working with paragraphs three and four of our sample document, as follows:

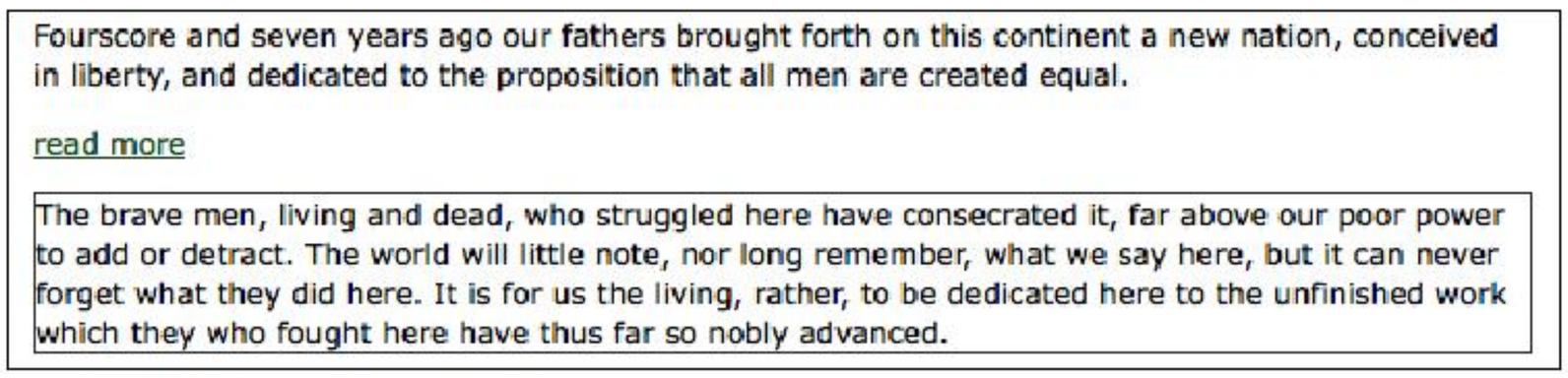
```
<p>Fourscore and seven years ago our fathers brought forth
  on this continent a new nation, conceived in liberty,
  and dedicated to the proposition that all men are
  created equal.</p>
<p>Now we are engaged in a great civil war, testing whether
  that nation, or any nation so conceived and so
  dedicated, can long endure. We are met on a great
  battlefield of that war. We have come to dedicate a
  portion of that field as a final resting-place for those
  who here gave their lives that the nation might live. It
  is altogether fitting and proper that we should do this.
  But, in a larger sense, we cannot dedicate, we cannot
  consecrate, we cannot hallow, this ground.</p>
<a href="#" class="more">read more</a>
<p>The brave men, living and dead, who struggled here have
  consecrated it, far above our poor power to add or
  detract. The world will little note, nor long remember,
  what we say here, but it can never forget what they did
  here. It is for us the living, rather, to be dedicated
  here to the unfinished work which they who fought here
  have thus far so nobly advanced.</p>
<p>It is rather for us to be here dedicated to the great
  task remaining before us&mdash;that from these honored
  dead we take increased devotion to that cause for which
  they gave the last full measure of devotion&mdash;that
  we here highly resolve that these dead shall not have
  died in vain&mdash;that this nation, under God, shall
  have a new birth of freedom and that government of the
  people, by the people, for the people, shall not perish
  from the earth.</p>
```


To help us see what's happening during the effect, we'll give the third paragraph a 1-pixel border and the fourth paragraph a gray background. We'll also hide the fourth paragraph when the DOM is ready, as shown in the following code snippet:

```
$(document).ready(function() {
  $('p').eq(2).css('border', '1px solid #333');
  $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

Listing 4.23

Our sample document now displays the opening paragraph, followed by the **read more** link and the bordered paragraph, as shown in the following screenshot:

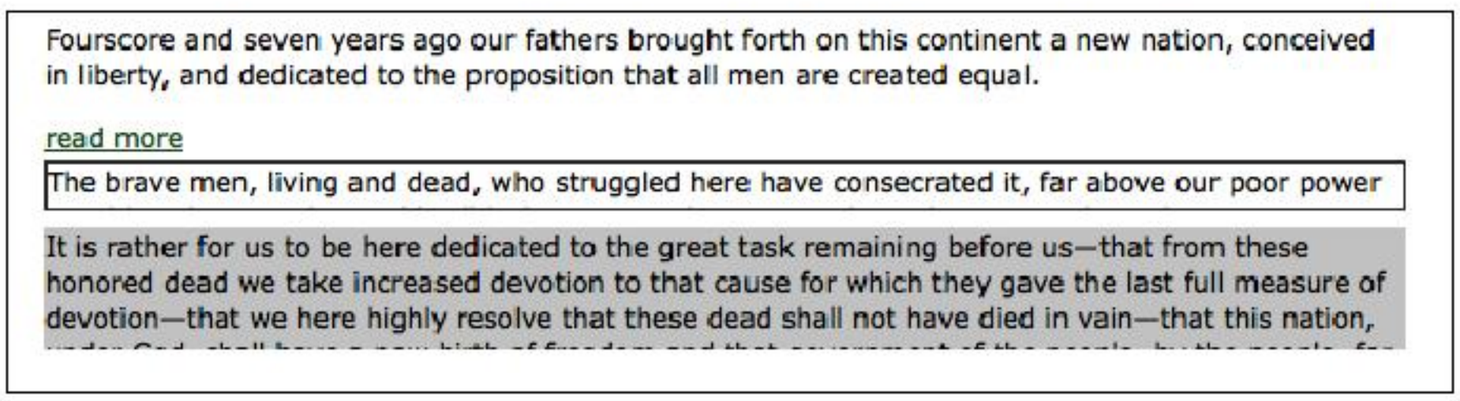


Finally, we'll add the `.click()` method to the third paragraph so that when it is clicked, the third paragraph will slide up (and out of view), while the fourth paragraph slides down (and into view):

```
$(document).ready(function() {
  $('p').eq(2)
    .css('border', '1px solid #333')
    .click(function() {
      $(this).slideUp('slow').next().slideDown('slow');
    });
  $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

Listing 4.24

A screenshot of these two effects in mid-slide confirms that they do, indeed, occur simultaneously, as follows:



The third paragraph, which started visible, is halfway through sliding up at the same time as the fourth paragraph, which started hidden, is halfway through sliding down.

Callbacks

In order to allow queuing effects on different elements, jQuery provides a **callback function** for each effect method. As we have seen with event handlers and with the `.queue()` method, callbacks are simply functions passed as method arguments. In the case of effects, they appear as the last argument of the method.

If we use a callback to queue the two slide effects, then we can have the fourth paragraph slide down before the third paragraph slides up. Let's first try moving the `.slideUp()` call into the `.slideDown()` method's completion callback, as shown in the following code snippet:

```
$(document).ready(function() {
  $('p').eq(2)
    .css('border', '1px solid #333')
    .click(function() {
      $(this).next().slideDown('slow', function() {
        $(this).slideUp('slow');
      });
    });
  $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

Listing 4.25

We do need to be careful here, however, about what is actually going to slide up. The **context** of the function—the keyword `this`—is different because the callback is inside the `.slideDown()` method. Here, `$(this)` is no longer the third paragraph, as it was directly within the `click` handler; rather, as the `.slideDown()` method is being called on the result of `$(this).next()`, the callback within that method now sees `$(this)` as the next sibling, or the fourth paragraph. Therefore, if we put `$(this).slideUp('slow')` inside the callback, as we have in Listing 4.25, we would end up hiding the same paragraph that we had just made visible.

A simple way to keep the `$(this)` reference stable is to store it in a variable right away within the `click` handler, like `var $clickedItem = $(this).`

Now `$clickedItem` will refer to the third paragraph, both outside and inside the effect method callback. The following code snippet is what the code looks like using our new variable:

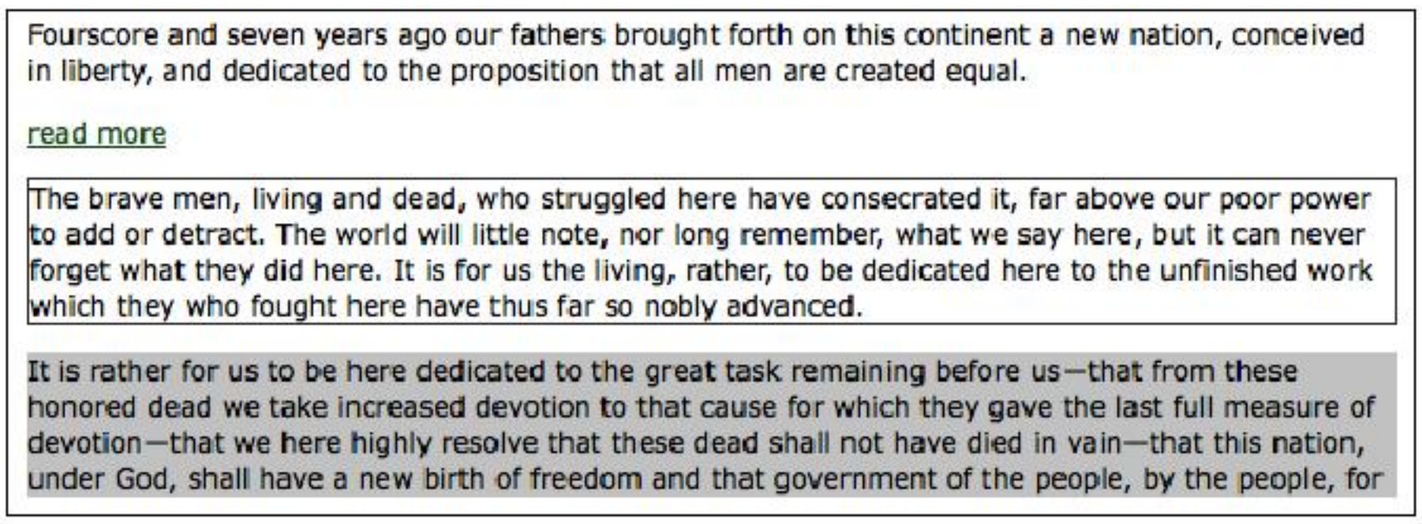
```
$(document).ready(function() {
  $('p').eq(2)
.css('border', '1px solid #333')
.click(function() {
  var $clickedItem = $(this);
```

```
        $clickedItem.next().slideDown('slow', function() {
         $clickedItem.slideUp('slow');
        });
      });
    $('p').eq(3).css('backgroundColor', '#ccc').hide();
});
```

Listing 4.26

Using `$clickedItem` inside the `.slideDown()` callback relies on the properties of **closures**. We'll be discussing this important, yet difficult-to-master, topic in *Appendix A*.

This time, a snapshot halfway through the effects will reveal that both the third and the fourth paragraphs are visible; the fourth has finished sliding down and the third is about to begin sliding up, as shown in the following screenshot:



Now that we've discussed callbacks, we can return to the code from Listing 4.22, in which we queued a background-color change near the end of a series of effects. Instead of using the `.queue()` method, as we did then, we can simply use a callback function, as follows:


```
$(document).ready(function() {
  $('div.label').click(function() {
    var paraWidth = $('div.speech p').outerWidth();
    var $switcher = $(this).parent();
    var switcherWidth = $switcher.outerWidth();
    $switcher
      .css({position: 'relative'})
      .fadeTo('fast', 0.5)
      .animate({
        left: paraWidth - switcherWidth
      }, {
        duration: 'slow',
```

```
        queue: false
      })
      .fadeTo('slow', 1.0)
      .slideUp('slow', function() {
        $switcher.css({backgroundColor: '#f00'});
      })
      .slideDown('slow');
});
```

Listing 4.27

Here, again, the background color of `<div id="switcher">` changes to red after it slides up, and before it slides back down. Note that when using an effect's completion callback rather than `.queue()`, we don't need to worry about calling `next()` from within the callback.

In a nutshell

With all the variations to consider when applying effects, it can become difficult to remember whether the effects will occur simultaneously or sequentially. A brief outline might help:

1. Effects on a single set of elements are:
    - **simultaneous** when applied as multiple properties in a single `.animate()` method
    - **queued** when applied in a chain of methods, unless the `queue` option is set to `false`
2. Effects on multiple sets of elements are:
    - **simultaneous** by default
    - **queued** when applied within the callback of another effect or within the callback of the `.queue()` method

Summary

By using effect methods that we have explored in this chapter, we should now be able to modify inline style attributes from JavaScript, apply pre-packaged jQuery effects to elements, and create our own custom animations. In particular, we learned to incrementally increase and decrease text size by using either the `.css()` or the `.animate()` method, gradually hide and show page elements by modifying several attributes, and to animate elements, simultaneously or sequentially, in a number of ways.

In the first four chapters of the book, all of our examples have involved manipulating elements that have been hard-coded into the page's HTML. In *Chapter 5*, we will explore ways to manipulate the DOM directly, including using jQuery to create new elements and insert them into the DOM wherever we choose.

Further reading

The topic of animation will be explored in more detail in *Chapter 11*. A complete list of available effect and styling methods is available in *Appendix C* of this book, in *jQuery Reference Guide*, or in the official jQuery documentation at `http://api.jquery.com/`.

Exercises

To complete these exercises, you will need the `index.html` file for this chapter, as well as the finished JavaScript code as found in `complete.js`. These files can be downloaded from the Packt Publishing website at `http://www.packtpub.com/support`.

The challenge exercise may require use of the official jQuery documentation at `http://api.jquery.com/`.

1. Alter the stylesheet to hide the contents of the page initially. When the page is loaded, fade in the contents slowly.
2. Give each paragraph a yellow background only when the mouse is over it.
3. Make a click of the title (`<h2>`) simultaneously make it fade to 25% opacity and get a left margin of `20px`, then when this is complete, fade the speech text to 50% opacity.
4. **Challenge**: React to presses of the arrow keys by smoothly moving the switcher box 20 pixels in the corresponding direction. The key codes for the arrow keys are: `37` (left), `38` (up), `39` (right), and `40` (down).