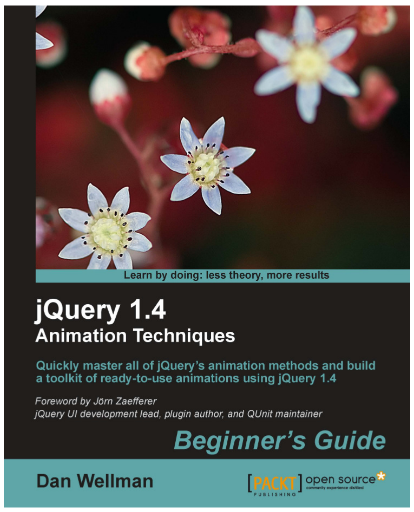
jQuery 1.4 Animation Techniques Beginner's Guide

Dan Wellman


Learn by doing: less theory, more results
jQuery 1.4
Animation Techniques
Quickly master all of jQuery's animation methods and build a toolkit of ready-to-use animations using jQuery 1.4
Foreword by Jörn Zaefferer
jQuery UI development lead, plugin author, and QUnit maintainer
Beginner's Guide
Dan Wellman
PACKT open source

Chapter No. 5
"Custom Animations"

In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.5 "Custom Animations"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Dan Wellman is an author and web developer based on the South coast of the UK. By day he works alongside some of the most talented people he has had the pleasure of calling colleagues, for a small, yet accomplished digital agency called Design Haus. By night he writes books and tutorials on a range of frontend topics. He is hopelessly addicted to jQuery. His life is enriched by four wonderful children, a beautiful wife, and a close circle of family and friends. This is his fifth book.

I would like to thank the hugely supportive and patient editorial team at Packt, without whom this book would not exist. I would also like to thank the reviewers, especially Ben Nadel and Cyril Pierron, who put aside their own personal projects and dedicated countless hours to ensuring the book's technical accuracy. I'd also like to say a big Hey! to some of my closest friends, in no particular order; Andrew Herman, Steev Bishop, Aaron Matheson, Eamon O'Donoghue, James Zabiela, Mike Woodford, and John Adams.

jQuery 1.4 Animation Techniques Beginner's Guide

jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML, and is the most popular JavaScript library in use today. Using the features offered by jQuery, developers are able to create dynamic web pages. This book will act as a resource for you to create animation and advanced special effects in your web applications, by following the easy-to-understand steps mentioned in it.

jQuery 1.4 Animation Techniques: Beginner's Guide will allow you to master animation in jQuery to produce slick and attractive interfaces that respond to your visitors' interactions. You will learn everything you need to know about creating engaging and effective web page animations using jQuery. The book uses many examples and explains how to create animations using an easy, step-by-step, beginner's guide approach.

This book provides various examples that gradually build up the reader's knowledge and practical experience in using the jQuery API to create stunning animations. The book starts off by explaining how animations make your user interface interactive and attractive. It explains the various methods used to make the element being animated appear or disappear. It provides a set of steps to create simple animations and show fading animations.

You can later learn how to make complex animations by chaining different effects together as well as how to halt a currently running application. You will find out how to slide your animation elements and learn to create custom animations that can be complex and specialized.

You will find out how to obtain and set up the jQuery UI—the official user interface library for jQuery. This book will tell you how to animate a page's background image, and will teach you how to make images scroll in a certain direction and at a certain speed depending on the movement of the mouse pointer.

What This Book Covers

Chapter 1, Introduction covers the basics including downloading jQuery and setting up a development area, a brief history of animation on the Web, when and where not to use animation, how animation can enhance an interface, and the animation methods exposed by jQuery. A basic example of animation is also covered.

Chapter 2, Fading Animations looks at the fading family of animation methods including fading elements in and out, fade toggling, triggering animations with show(), hide(), and toggle(), and fading an element to a specific opacity.

Chapter 3, Managing Animations covers the animation queue and the methods jQuery provides for managing it. We see how to clear the queue, how to add functions to it, and how to clear it. We see how to add a delay between queued items and how to prevent animations building up in the queue when they are not required.

Chapter 4, Sliding Animations looks at jQuery's sliding animation and covers how to slide elements in an out of view and how to toggle the slide based on their current state. We also look at how CSS positioning can affect animations and how to avoid a common pitfall when using these methods in a drop-down menu.

Chapter 5, Custom Animations focuses on the animate() method, which jQuery provides for us as a means of creating custom animations not already predefined. This extremely powerful method allows us to animate almost any CSS-style property to easily create complex and attractive animations.

Chapter 6, Extended Animations with jQuery UI looks at the additional effects added by jQuery UI, the official UI library built on top of jQuery. We look at each of the 14 new effects as well as covering the easing functions built into the library.

Chapter 7, Full Page Animations looks at animations that form the main focus of the page. Techniques we cover include animating page scroll, creating a parallax effect, and creating basic stop-motion animations.

Chapter 8, Other Popular Animations looks at some common types of animations found on the web including proximity animations triggered by the mouse pointer, animated headers, and a modern-day equivalent to the marquee element.

Chapter 9, CSS3 Animations covers how we can use CSS3 to create attractive animations driven by the latest CSS transforms and how jQuery can be used to make the process easier, including the latest cssHooks functionality.

Chapter 10, Canvas Animations looks at the HTML5 canvas element and shows how it can be used to create stunning animations without the use of Flash or other proprietary technologies. The book closes with an in-depth example teaching how to create an interactive game using nothing but HTML and JavaScript.

5
Custom Animations

The predefined effects that we have looked at throughout the book so far are very good at what they do, but they are there to cater for very specific requirements and will sometimes not be enough when more complex animations are needed.

In these situations we can use jQuery's `animate()` *method, which allows us to define custom animations with ease that can be as complex and as specialized as the task at hand requires, and this is what we'll be looking at over the course of this chapter.*

Subjects that we'll cover throughout the course of this chapter will include:

- Creating custom animations with the `animate()` method
- Passing arguments to the method
- Animating an element's dimensions
- Animating an element's position
- Creating a jQuery animation plugin

The animate method

All custom animations with jQuery are driven with the `animate()` method. Despite the ability to animate almost any style property that has a numeric value, the method is simple to use and takes just a few arguments. The method may be used in the following way:

```
jQuery(elements).animate(properties to animate,
  [duration],
  [easing],
  [callback]
);
```

The first argument should take the form of an object where each property of the object is a style that we'd like to animate, very similar to how we would use jQuery's `css()` method.

As I mentioned before, this can be any CSS style that takes a purely numerical argument (with the exception of colors, although with the jQuery UI library we can animate colors as well. See *Chapter 6*, *Extended Animations with jQuery UI* for more information on jQuery UI). Background positions cannot be animated by jQuery natively, but it is quite easy to animate this property manually; see *Chapter 7*, *Full Page Animations* for more information on this technique.

The duration, easing, and callback arguments take the same formats as those that we used with the fading and sliding methods earlier in the book and are used in exactly the same way.

Per-property easing

As of the 1.4 version of jQuery, we can apply different types of easing to each style property we are animating when using the `animate()` method. So if we are animating both the `width` and `height` of an element for example, we can use `linear` easing for the `width` animation, and `swing` easing for the `height` animation. This applies to the standard easing functions built into jQuery, or any of the easing functions added with the easing plugin that we looked at in *Chapter 4*, *Sliding Animations*.

To supply easing types to the `animate()` method on a per-property basis, we need to provide an array as the value of the property we are animating. This can be done using the following syntax:

```
jQuery(elements).animate({
  property: [value, easingType]
});
```

An alternative syntax for animate()

Instead of using the duration, easing, and callback arguments individually, we may alternatively pass a configuration object to the `animate()` method containing the following configuration options:

- `duration`
- `easing`
- `complete`
- `step`
- `queue`
- `specialEasing`

The first three options are the same as the arguments would be if we passed them into the method in the standard way. The last three are interesting however, in that we do not have access to them in any other way.

The `step` option allows us to specify a callback function that will be executed on each step of the animation. The `queue` option accepts a Boolean that controls whether the animation is executed immediately or placed into the selected element's queue. The `specialEasing` option allows us to specify an easing function for each individual style property that is being animated, giving us easing on a per-property basis using the alternative syntax.

The pattern for this second method of usage is as follows:

```
jQuery(elements).animate(properties to animate, [configuration
  options]);
```

Like most (but not all) jQuery methods, the `animate()` method returns a jQuery object so that additional methods can be chained to it. Like the other effect methods, multiple calls to `animate()` on the same element will result in an animation queue being created for the element. If we want to animate two different style properties at the same time, we can pass all required properties within the object passed to the `animate()` method as the first argument.

Animating an element's position

The `animate()` method is able to animate changes made to any CSS style property that has a numeric value, with the exception of colors and `background-positions`. In this example, we'll create a content viewer that shows different panels of content by sliding them in and out of view using the `animate()` method.

This type of widget is commonly used on portfolio or showcase sites and is an attractive way to show a lot of content without cluttering a single page. In this example, we'll be animating the element's position.

Time for action – creating an animated content viewer

We'll start again by adding the underlying markup and styling.

1. The underlying markup for the content viewer should be as follows:

```
<div id="slider">
  <div id="viewer">
    <img id="image1" src="img/amstrad.jpg" alt="Amstrad CPC 472">
    <img id="image2" src="img/atari.jpg" alt="Atari TT030">
    <img id="image3" src="img/commodore16.jpg" alt="Commodore 64">
    <img id="image4" src="img/commodore128.jpg" alt="Commodore
      128">
    <img id="image5" src="img/spectrum.jpg" alt="Sinclair ZX
      Spectrum +2">
  </div>
  <ul id="ui">
    <li class="hidden" id="prev">
      <a href="" title="Previous">&laquo;</a></li>
    <li><a href="#image1" title="Image 1" class="active">Image
      1</a></li>
    <li><a href="#image2" title="Image 2">Image 2</a></li>
    <li><a href="#image3" title="Image 3">Image 3</a></li>
    <li><a href="#image4" title="Image 4">Image 4</a></li>
    <li><a href="#image5" title="Image 5">Image 5</a></li>
    <li class="hidden" id="next">
      <a href="" title="Next">&raquo;</a></li>
  </ul>
</div>
```

2. Save the file as `animate-position.html`.

3. Next we should create the base CSS. By that I mean that we should add the CSS which is essential for the content-viewer to function as intended, as opposed to styling that gives the widget a theme or skin. It's good practice to separate out the styling in this way when creating plugins so that the widget is compatible with jQuery UI's Themeroller theming mechanism.

4. In a new file in your text editor add the following code:

```
#slider { width:500px; position:relative; }
#viewer {
  width:400px; height:300px; margin:auto; position:relative;
  overflow:hidden;
}
#slider ul {
  width:295px; margin:0 auto; padding:0; list-style-type:none;
}
#slider ul:after {
  content:"."; visibility:hidden; display:block; height:0;
  clear:both;
}
#slider li { margin-right:10px; float:left; }
#prev, #next { position:absolute; top:175px; }
#prev { left:20px; }
#next { position:absolute; right:10px; }
.hidden { display:none; }
#slide {
  width:2000px; height:300px; position:absolute; top:0; left:0;
}
#slide img { float:left; }
#title { margin:0; text-align:center; }
```

5. Save this in the `css` folder as `animate-position.css`, and don't forget to link to the new stylesheet from the `<head>` of our page. Run the page in your browser now, before we get into the scripting, so that you can see how the widget behaves without the accompanying script. You should find that any image can be viewed by clicking its corresponding link using only CSS, and this will work in any browser. The previous and next arrows are hidden with our CSS because these will simply not work with JS turned off and the image titles are not displayed, but the widget's core functionality is still fully accessible.

What just happened?

The underlying HTML in this example is very straightforward. We have an outer container for the content-viewer as a whole, then within this we have a container for our content panels (simple images in this example) and a navigation structure to allow the different panels to be viewed.

Some of the elements we've added style rules for in our CSS file aren't hardcoded into the underlying markup, but will be created as necessary when needed. Doing it this way ensures that the content-viewer is still usable even when the visitor has JavaScript disabled.

One important point to note is that the `#slide` wrapper element that we create and wrap around the images has a `height` equal to a single image and a `width` equal to the sum of all image widths. The `#viewer` element on the other hand has both a `width` and a `height` equal to a single image so that only one image is visible at any one time.

With JavaScript disabled, the images will appear to stack up on top of each other, but once the `#slide` wrapper element has been created the images are set to float in order to stack up horizontally.

We'll use easing in this example, so be sure to link to the easing plugin directly after the jQuery reference at the end of the `<body>`:

```
<script src="js/jquery.easing.1.3.js"></script>
```

Time for action – initializing variables and prepping the widget

First we need to prepare the underlying markup and store some element selectors:

```
$("#viewer").wrapInner("<div id=\"slide\"></div>");

var container = $("#slider"),
  prev = container.find("#prev"),
  prevChild = prev.find("a"),
  next = container.find("#next").removeClass("hidden"),
  nextChild = next.find("a"),
  slide = container.find("#slide"),
  key = "image1",
  details = {
    image1: {
      position: 0, title: slide.children().eq(0).attr("alt")
    },
    image2: {
      position: -400, title: slide.children().eq(1).attr("alt")
    },
    image3: {
      position: -800, title: slide.children().eq(2).attr("alt")
    },
    image4: {
      position: -1200, title: slide.children().eq(3).attr("alt")
    },
    image5: {
      position: -1600, title: slide.children().eq(4).attr("alt")
    }
  };

$("<h2>", {
  id: "title",
  text: details[key].title
}).prependTo("#slider");
```

What just happened?

To start with, we first wrap all of the images inside the `#viewer <div>` in a new container. We'll be using this container to animate the movement of the panels. We give this new container an `id` attribute so that we can easily select it from the DOM when required.

This is the element that we will be animating later in the example.

Next we cache the selectors for some of the elements that we'll need to manipulate frequently. We create a single jQuery object pointing to the outer `#slider` container and then select all of the elements we want to cache, such as the previous and next arrows, using the jQuery `find()` method.

A `key` variable is also initialized which will be used to keep track of the panel currently being displayed. Finally, we create a `details` object that contains information about each image in the content viewer. We can store the `left` position in pixels that the `slide` container must be animated to in order to show any given panel, and we can also store the title of each content panel.

The title of each panel is read from the `alt` attribute of each image, but if we were using other elements, we could select the `title` attribute, or use jQuery's data method to set and retrieve the title of the content.

The `<h2>` element used for the title is created and inserted into the content-viewer with JS because there is no way for us to change it without using JS. Therefore when visitors have JS disabled, the title is useless and is better off not being shown at all.

The last thing we do in the first section of code is to remove the `hidden` class name from the next button so that it is displayed.

The previous link (by this I mean the link that allows the visitor to move to the previous image in the sequence) is not shown initially because the first content panel is always the panel that is visible when the page loads, so there are no previous panels to move to.

Time for action – defining a post-animation callback

Next we need a function that we can execute each time an animation ends:

```
function postAnim(dir) {

  var keyMatch = parseInt(key.match(/\d+$/));

  (parseInt(slide.css("left")) < 0) ? prev.show() : prev.hide();

  (parseInt(slide.css("left")) === -1600) ? next.hide() :
    next.show();
```

```
      if (dir) {
        var titleKey = (dir === "back") ? keyMatch - 1 : keyMatch + 1;
        key = "image" + titleKey;
      }

      container.find("#title").text(details[key].title);

      container.find(".active").removeClass("active");
      container.find("a[href=#" + key + "]").addClass("active");
    };
```

What just happened?

In this second section of code, we define a function that we'll call after an animation ends. This is used for some housekeeping to do various things that may need doing repeatedly, so it is more efficient to bundle them up into a single function instead of defining them separately within event handlers. This is the `postAnim()` function and it may accept a single parameter which refers to the direction that the slider has moved in.

The first thing we do in this function is use the regular expression `/\d+$/` with JavaScript's `match()` function to parse the panel number from the end of the string saved in the `key` variable which we initialized in the first section of code, and which will always refer to the currently visible panel.

Our `postAnim()` function may be called either when a panel is selected using the numeric links, or when the previous/next links are used. However, when the previous/next links are used we need the `key` to know which panel is currently being displayed in order to move to the next or previous panel.

We then check whether the first panel is currently being displayed by checking the `left` CSS style property of the `#slide` element. If the `#slide` element is at `0`, we know the first panel is visible so we hide the previous link. If the `left` property is less than `0`, we show the previous link. We do a similar test to check whether the last panel is visible, and if so, we hide the next link. The previous and next links will only be shown if they are currently hidden.

We then check whether the `dir` (direction) argument has been supplied to the function. If it has, we have to work out which panel is now being displayed by reading the `keyMatch` variable that we created earlier and then either subtracting `1` from it if the `dir` argument is equal to `back`, or adding `1` to it if not.

The result is saved back to the `key` variable, which is then used to update the `<h2>` title element. The title text for the current panel is obtained from our `details` object using the `key` variable. Lastly we add the class name `active` to the numeric link corresponding to the visible panel.

Although not essential, this is something we will want to use when we come to add a skin to the widget. We select the right link using an attribute selector that matches the `href` of the current link. Note that we don't create any new jQuery objects in this function; we use our cached `container` object and the `find()` method to obtain the elements we require.

Time for action – adding event handlers for the UI elements

Now that the slider has been created, we can add the event handlers that will drive the functionality:

```
$("#ui li a").not(prevChild).not(nextChild).click(function(e){
  e.preventDefault();

  key = $(this).attr("href").split("#")[1];

  slide.animate({
    left: details[key].position
  }, "slow", "easeOutBack", postAnim);
});

nextChild.add(prevChild).click(function(e){
  e.preventDefault();

  var arrow = $(this).parent();

  if (!slide.is(":animated")) {
    slide.animate({
      left: (arrow.attr("id") === "prev") ? "+=400" : "-=400"
    }, "slow", "easeOutBack", function(){

      (arrow.attr("id") === "prev") ? postAnim("back") :
        postAnim("forward")
    });
  }
});
```

What just happened?

The first handler is bound to the main links used to display the different panels, excluding the previous and next links with the jQuery `not()` method. We first stop the browser following the link with the `preventDefault()` method.

We then update the `key` variable with the panel that is being displayed by extracting the panel name from the link's `href` attribute. We use JavaScript's `split()` method to obtain just the panel `id` and not the `#` symbol.

Finally, we animate the slide element by setting its `left` CSS style property to the value extracted from the `details` object. We use the `key` variable to access the value of the `position` property.

As part of the animation, we configure the duration as `slow` and the easing as `easeOutBack`, and specify our `postAnim` function as the callback function to execute when the animation ends.

Finally, we need to add a click handler for the previous/next links used to navigate to the next or previous image. These two links can both share a single click handler. We can select both of these two links using our cached selectors from earlier, along with jQuery's `add()` method to add them both to a single jQuery object in order to attach the handler functions to both links.

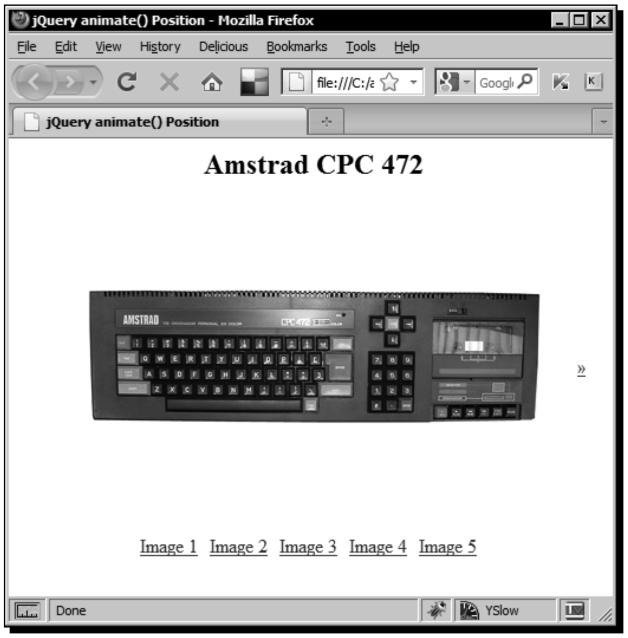
We again stop the browser from following the link using `preventDefault()`. We then cache a reference to the parent of the link that was clicked, using the `arrow` variable, so that we can easily refer to it later on in the function. This is needed because within the callback function for the `animate()` method, the `$(this)` keyword will be scoped to the `#slide` element instead of the link that was clicked.

We then check that the `#slide` element is not already being animated using the `:animated` filter. This check is important because it prevents the viewer breaking if one of the links is clicked repeatedly.

If it is not already being animated, we perform the animation and move the slide element either `400` pixels (the `width` of a single content panel) backwards or forwards. We can check which arrow was clicked by looking at the `id` attribute of the element referenced by the `arrow` variable.

We specify the same duration and easing values as before in the animation method, but instead of passing a reference to the `postAnim` function as the callback parameter we pass an anonymous function instead. Within this anonymous function, we determine which link was clicked and then call the `postAnim` function with the appropriate argument. Remember, this is necessary to obtain the correct key for the `details` object because neither the previous nor the next links have `href` attributes pointing to an image.

Try the page out in a browser at this point and you should find that an image can be viewed by clicking on any of the links, including the previous and next links. This is how the widget should appear at this stage:


jQuery animate() Position - Mozilla Firefox
File Edit View History Delicious Bookmarks Tools Help
file:///C:/
Google
jQuery animate() Position
Amstrad CPC 472
»
Image 1 Image 2 Image 3 Image 4 Image 5
Done
YSlow

The previous screenshot shows the widget in its un-skinned state, with only the CSS required for it to function included.

Skinning the widget

'There's more than one way to skin a cat' was once proclaimed, and this applies to widgets as well as cats. Lastly, let's add some custom styling to the widget to see how easy it is to make the widget attractive as well as functional.

Time for action – adding a new skin

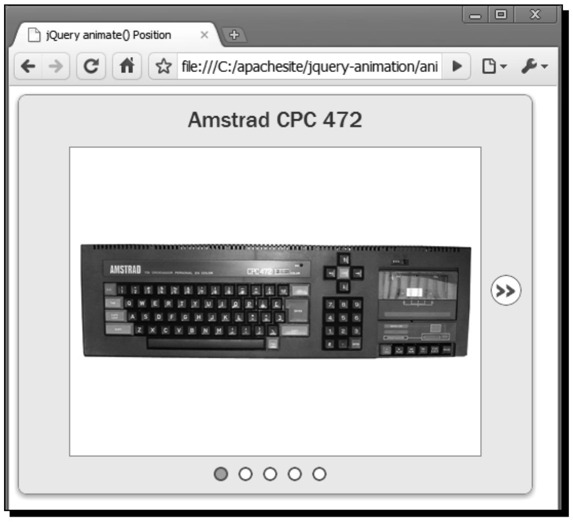
At the bottom of the `animate-position.css` file, add the following code:

```
a { outline:0 none; }
#slider {
  border:1px solid #999; -moz-border-radius:8px;
  -webkit-border-radius:8px; border-radius:8px;
  background-color:#ededed; -moz-box-shadow:0px 2px 7px #aaa;
  -webkit-box-shadow:0px 2px 7px #aaa; box-shadow:0px 2px 7px #aaa;
}
#title, #slider ul { margin-top:10px; margin-bottom:12px; }
#title {
  font:normal 22px "Nimbus Sans L", "Helvetica Neue", "Franklin
    Gothic Medium", Sans-serif;
  color:#444;
}
#viewer { border:1px solid #999; background-color:#fff; }
#slider ul { width:120px; }
#slider ul li a {
  display:block; width:10px; height:10px; text-indent:-5000px;
  text-decoration:none; border:2px solid #666;
  -moz-border-radius:17px; -webkit-border-radius:17px;
  border-radius:17px; background-color:#fff; text-align:center;
}
#slider #prev, #slider #next { margin:0; text-align:center; }
#slider #prev { left:10px; }
#slider #prev a, #slider #next a {
  display:block; height:28px; width:28px; line-height:22px;
  text-indent:0; border:1px solid #666; -moz-border-radius:17px;
  -webkit-border-radius:17px; border-radius:17px;
  background-color:#fff;
}
#prev a, #next a { font:bold 40px "Trebuchet MS"; color:#666; }
#slider ul li a.active { background-color:#F93; }
```

What just happened?

With this code we style all of the visual aspects of the widget without interfering with anything that controls how it works. We give it some nice rounded corners and add a drop-shadow to the widget, turn the numeric links into little clickable icons, and style the previous and next links. Colors and fonts are also set in this section as they too are obviously highly dependent on the theme.

These styles add a basic, neutral theme to the widget, as shown in the following screenshot:


jQuery animate() Position
file:///C:/apachesite/jquery-animation/ani
Amstrad CPC 472

The styles we used to create the theme are purely arbitrary and simply for the purpose of the example. They can be changed to whatever we need in any given implementation to suit other elements on the page, or the overall theme of the site.

Pop quiz – creating an animated content-viewer

1. What arguments may the `animate()` method be passed?
   a. An array where the array items are the element to animate, the duration, the easing, and a callback function
   b. The first argument is an object containing the style properties to animate, optionally followed by the duration, an easing type, and a callback function
   c. An object where each property refers to the style properties to animate, the duration, easing, and a callback function
   d. A function which must return the style properties to animate, the duration, easing, and a callback function
2. What does the `animate()` method return?
   a. An array containing the style properties that were animated
   b. A array containing the elements that were animated
   c. A jQuery object for chaining purposes
   d. A Boolean indicating whether the animation completed successfully

Have a go hero – making the image viewer more scalable

In our animated content viewer, we had a fixed number of images and a hardcoded navigation structure to access them. Extend the content viewer so that it will work with an indeterminate number of images. To do this, you will need to complete the following tasks:

- Determine the number of images in the content viewer at run time and set the `width` of the `#slide` wrapper element based on the number of images
- Build the navigation links dynamically based on the number of images
- Create the `details` object dynamically based on the number of images and set the correct `left` properties to show each image

Animating an element's size

As I mentioned at the start of the chapter, almost any style property that contains a purely numeric value may be animated with the `animate()` method.

We looked at animating an element's position by manipulating its `left` style property, so let's move on to look at animating an element's size by manipulating its `height` and `width` style properties.

In this example, we'll create image wrappers that can be used to display larger versions of any images on the page by manipulating the element's size.

Time for action – creating the underlying page and basic styling

First, we'll create the underlying page on which the example will run.

1. Add the following HTML to the `<body>` of our template file:

```
<article>
  <h1>The Article Title</h1>
  <p><img id="image1-thumb" class="expander" alt="An ASCII Zebra"
    src="img/ascii.gif" width="150" height="100">Lorem ipsum
    dolor...</p>
  <p><img id="image2-thumb" class="expander" alt="An ASCII Zebra"
    src="img/ascii2.gif" width="100" height="100">Lorem ipsum
    dolor...</p>
</article>
```

2. Save the example page as `animate-size.html`. We'll keep the styling light in this example; in a new file in your text editor, add the following code:

```
article {
  display:block; width:800px; margin:auto; z-index:0;
  font:normal 18px "Nimbus Sans L", "Helvetica Neue", "Franklin
    Gothic Medium", sans-serif;
}
article p {
  margin:0 0 20px; width:800px; font:15px Verdana, sans-serif;
  line-height:20px;
}
article p #image2-thumb { float:right; margin:6px 0 0 30px; }
img.expander { margin:6px 30px 1px 0; float:left; }
.expander-wrapper { position:absolute; z-index:999; }
.expander-wrapper img {
  cursor:pointer; margin:0; position:absolute;
}
.expander-wrapper .expanded { z-index:9999; }
```

3. Save this file as `animate-size.css` in the `css` folder.

What just happened?

The HTML could be any simple blog post consisting of some text and a couple of images. The points to note are that each image is given an `id` attribute so that it can be easily referenced, and that each image is actually the full-sized version of the image, scaled down with `width` and `height` attributes.

The styles used are purely to lay out the example; very little of the code is actually required to make the example work. The `expander-wrapper` styles are needed to position the overlaid images correctly, but other than that the styling is purely arbitrary.

We're floating the second image to the right. Again this isn't strictly necessary; it's used just to make the example a little more interesting.

Time for action – defining the full and small sizes of the images

First we need to specify the full and small sizes of each image:

```
var dims = {
  image1: {
    small: { width: 150, height: 100 },
    big: { width: 600, height: 400 }
  },
  image2: {
    small: { width: 100, height: 100 },
    big: { width: 400, height: 400 }
  }
},
webkit = ($("body").css("-webkit-appearance") !== "" && $("body").
css("-webkit-appearance") !== undefined) ? true : false;
```

What just happened?

We create an object which itself contains properties matching each image's filename. Each property contains another nested object which has `small` and `big` properties and the relevant integers as values. This is a convenient way to store structured information that can easily be accessed at different points in our script.

We also create a variable called `webkit`. There is a slight bug in how images floated to the right are treated in Webkit-based browsers such as Safari or Chrome. This variable will hold a Boolean that will indicate whether Webkit is in use.

A test is performed which tries to read the `-webkit-appearance` CSS property. In Webkit browsers, the test will return `none` as the property is not set, but other browsers will either return an empty string or the value `undefined`.

Time for action – creating the overlay images

Next we should create an almost exact copy of each image on the page to use as an overlay:

```
$(".expander").each(function(i) {

  var expander = $(this),
    coords = expander.offset(),
    copy = $("<img>", {
      id: expander.attr("id").split("-")[0],
      src: expander.attr("src"),
      width: expander.width(),
      height: expander.height()
    });
```

What just happened?

In this part of the `<script>`, we select each image on the page and process them using jQuery's `each()` method. We set some variables, caching a reference to the current image and storing its coordinates on the page relative to the document using the jQuery `offset()` method.

We then create a new image for each existing image on the page, giving it an `id` attribute that pairs it with the image it is overlaying, the `src` of the original image, and the `width` and `height` of the original image. We use the JavaScript `split()` function to remove the part of the string that says `thumb` when we set the `id` of the new image.

Note that the previous code does not represent an entire snippet of fully-functional code. The outer function passed to the `each()` method has not yet been closed as we need to add some additional code after these variables.

Time for action – creating the overlay wrappers

We now need to create the wrappers for each of the overlay images (note that this code is still within the `each()` method and so will be executed for each of the images that have the `expanded` class name):

```
  $("<div></div>", {
    "class": "expander-wrapper",
    css: {
      top: coords.top,
      left: (webkit === true && expander.css("float") === "right") ?
        (coords.left + expander.width()) : coords.left,
        direction: (expander.css("float") === "right") ? "rtl" :
        "ltr"
    },
    html: copy,
    width: expander.width(),
    height: expander.height(),
    click: function() {

      var img = $(this).find("img"),
        id = img.attr("id");

      if (!img.hasClass("expanded")) {
        img.addClass("expanded").animate({
          width: dims[id].big.width,
          height: dims[id].big.height
        }, {
          queue: false
        });
      } else {
        img.animate({
          width: dims[id].small.width,
          height: dims[id].small.height
        }, {
          queue: false,
          complete: function() {
            $(this).removeClass("expanded");
          }
        });
      }
    }
  }).appendTo("body");
```

What just happened?

In this section of code, we create the wrapper element for the new image. We give it a new class name so that it can be positioned correctly.

Quoting the class property

We need to use quotes around the property name `class` so that it works correctly in Internet Explorer. If we fail to quote it, IE will throw a script error stating that it **expected an identifier, string, or number**.

We set the position of the wrapper element using the `css` property in conjunction with the coordinates we obtained from the `offset()` method earlier.

When setting the `left` position of the wrapper element, we need to check our `webkit` variable to see if Safari is in use. If this variable is set to `true`, and if the image is floated to the right, we position the overlay according to the `cords.left` value in addition to the `width` of the original image. If the `webkit` variable is `false`, or if the original image is floated `left`, we just set the `left` position of the wrapper to the value stored in `coords.left`.

We also need to set the `direction` property of any images that are floated right. We check the `float` style property and set the `direction` to `rtl` if the image is floated right, or `ltr` if not. This is done using JavaScript's ternary conditional.

This check is done so that the wrapper expands from right-to-left when the image is floated `right`. If we didn't set this, the wrapper would open up from left-to-right, which could make the full-sized image overflow the viewport or the content container resulting in scroll bars.

We add the new image to the wrapper by passing a reference to it into the jQuery `html()` method, and set the `width` of the wrapper to the `width` of the original (and new) image. This is necessary for the overlay to be positioned correctly over any images that are floated right.

Next we add a click handler to the wrapper. Within the anonymous function passed as the value of the `click()` method, we first cache a reference to the image within the wrapper that was clicked, and get the `id` of the image for convenience. Remember, the `id` of the overlay image will be the same as the original image it is covering minus the text string `-thumb`.

We then check whether the image has the class name `expanded`. If it doesn't, we add the class name and then animate the image to its full size using the second format of the `animate()` method. We pass two objects into the method as arguments; the first contains the CSS properties we wish to animate, in this case the `width` and `height` of the image.

The correct `width` and `height` to increase the image to are retrieved from the `dims` object using the `id` of the image that was clicked as the key. In the second object passed to the `animate()` method, we set the `queue` property to `false`. This has the same effect as using the `stop()` method directly before the `animate()` method and ensures that nothing bad happens if the overlay wrapper is repeatedly clicked.

If the image already has the class name `expanded`, we animate the image back to its small size. Again we use the two-object format of the `animate()` method, supplying `false` as the value of the `queue` property, and removing the class name `expanded` in an anonymous callback function passed to the `complete` property. Once the wrapper has been created, we append it to the `<body>` of the page.

At this point the code we've written will work as intended—clicking an image will result in the expanded version being animated to its full size. However, if the page is resized at all, the overlays will no longer be overlaying their images.

Time for action – maintaining the overlay positions

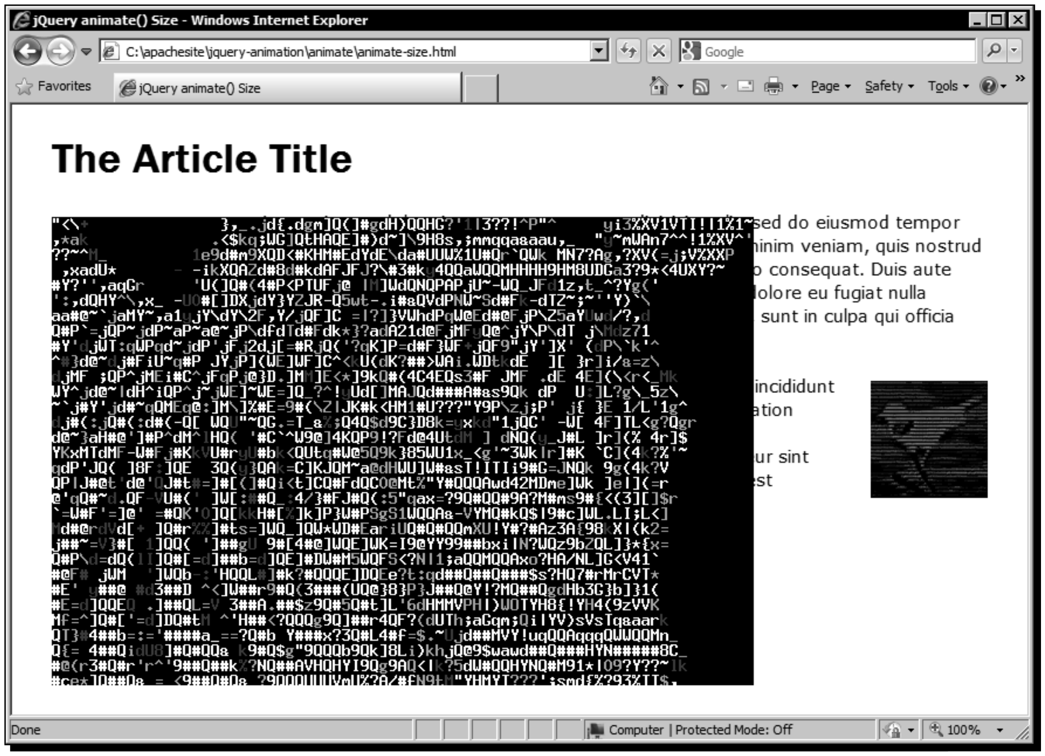
Because the overlays are positioned absolutely, we need to prevent them from becoming misaligned if the window is resized:

```
$(window).resize(function() {

  $("div.expander-wrapper").each(function(i) {

    var newCoords = $("#image" + (i + 1) + "-thumb").offset();

    $(this).css({
      top: newCoords.top,
      left: newCoords.left
    });
  });
});
```

What just happened?

All we need to do is make sure the overlay images stay directly on top of the original images when the page resizes, which we can achieve by binding a handler for the resize event to the `window` object. In the handler function, we just get the new coordinates of the underlying image, and set the `top` and `left` properties of the wrapper accordingly. Note that we don't animate the repositioning of the overlays.

Save the file and preview it in your browser. We should find that we can click on either image and it will expand to show a full-sized version of the image, with the first image expanding to the right, and the second expanding to the left:


jQuery animate() Size - Windows Internet Explorer
C:\apachesite\jquery-animation\animate\animate-size.html
Google
Favorites
jQuery animate() Size
Page
Safety
Tools
Done
Computer | Protected Mode: Off
100%

The Article Title

sed do eiusmod tempor
inim veniam, quis nostrud
o consequat. Duis aute
lolore eu fugiat nulla
sunt in culpa qui officia

incididunt
ation

ur sint
st

In the previous screenshot we see the first image as it expands to its full size.

Pop quiz – creating expanding images

1. In this example, we used a different format for the arguments passed to the `animate()` method, what format did the arguments take?
   a. Two arrays where the first array contains selectors for the elements to animate, and the second contains the duration, easing, and `specialEasing` strings, and a callback function
   b. A single object containing the style properties to animate, duration, easing, and `specialEasing` strings, and `step` and `complete` callback functions
   c. A function which must return the style properties to animate, the duration and easing strings, and a callback function

   d. Two objects where the first object contains the style properties to animate, and the second object contains the duration, easing and `specialEasing` strings, a Boolean indicating whether to queue repeated `animate()` calls, and the step and complete callback functions

2. What is the keyword `this` scoped to in an animation's callback function?
   a. The element that was animated
   b. The current window
   c. The container of the element that was animated
   d. The event object

Have a go hero – doing away with the hardcoded dims object

In the previous example, we hardcoded an image into the top of our script that was used to tell the `animate()` method what size the image should be animated to. While this was fine for the purpose of the example, it doesn't really scale well as a long-term solution as we would have to remember to set this every time we used the script (or otherwise ensure our images are always a fixed size).

The problem is that we have no way to programmatically get both the full size and thumb size from a single image. The good news is that any data that can be stored in a JavaScript object can also be passed across a network for consumption as a JSON object. Extend this example so that when the page loads, it passes the `src` attributes of the images on the page to the server, which returns a JSON object containing the small and large image sizes. An image manipulation library, like GD or ImageMagick, for PHP, or the `System.Drawing.Image` type in .Net, will be your friend here.

Creating a jQuery animation plugin

Plugins are an excellent way of packaging up functionality into an easy to deploy and share module of code that serves a specific purpose. jQuery provides the `fn.extend()` method precisely for this purpose, making it easy to create powerful and effective plugins that can be easily distributed and used.

There are a few guidelines that should be adhered to when creating jQuery plugins; these are as follows:

- New methods, which are called like other jQuery methods, for example `$(elements).newMethod()` should be attached to the `fn` object, and new functions, which are used by the plugin, for example `$.myFunction()`, should be attached to the `jQuery` object

- New methods and functions should always end in a semi-colon (`;`) to preserve functionality when the plugin is compressed
- Inside methods, the `this` keyword always refers to the current selection of elements, and methods should always return `this` to preserve chaining
- Always attach new methods and functions to the `jQuery` object as opposed to the `$` alias, unless using an anonymous function with an aliased `$` object

In this section, we'll create a plugin which can be used to create advanced transition effects when showing a series of images. The finished widget will be similar in some respects to the image viewer we created earlier, but will not animate the images themselves. Instead, it will apply transition effects between showing them.

Time for action – creating a test page and adding some styling

Once again we'll create the example page and basic styling first and add the script last.

1. The underlying HTML for this example is very light. All we need in the `<body>` of our template file are the following elements:

```
<div id="frame">
    <img class="visible" src="img/F-35_Lightning.jpg" alt="F-35
      Lightning">
    <img src="img/A-12_Blackbird.jpg" alt="A-12 Blackbird">
    <img src="img/B-2_Spirit.jpg" alt="B-2 Spirit">
    <img src="img/SR-71_Blackbird.jpg" alt="SR-71 Blackbird">
    <img src="img/F-117_Nighthawk.jpg" alt="F-117 Nighthawk">
</div>
```

2. Save this page as `advanced-transitions.html`.

3. Like the markup, the CSS we rely on for a plugin should also be as minimal as possible. Luckily not much CSS is required for our small collection of elements.

4. Add the following code to a new file in your text editor:

```
#frame { position:relative; width:520px; height:400px; z-index:0;
}
#frame img { position:absolute; top:0; left:0; z-index:1; }
#frame img.visible { z-index:2; }
#frame a {
  display:block; width:50%; height:100%; position:absolute; top:0;
  z-index:10; color:transparent;
  background-image:url(transparent.gif); filter:alpha(
    opacity = 0);
  text-align:center; text-decoration:none;
  font:90px "Palatino Linotype", "Book Antiqua", Palatino, serif;
  line-height:400%;
}
#frame a:hover {
  color:#fff; text-shadow:0 0 5px #000; filter:alpha(
    opacity = 100);
  filter: Shadow(Color=#000, Direction=0);
}
#frame a:focus { outline:none; }
#prev { left:0; }
#next { right:0; }
#overlay {
  width:100%; height:100%; position:absolute; left:0; top:0;
  z-index:3;
}
#overlay div { position:absolute; }
```

5. Save this in the `css` folder as `advanced-transitions.css`.

What just happened?

All we have on the underlying page are the images we wish to transition between within a container. It's best to keep the markup requirements for plugins as simple as possible so that they are easy for others to use and don't place undue restrictions on the elements or structure they want to use.

The images are positioned absolutely within the container using CSS so that they stack up on top of one another, and we set our `visible` class on the first element to ensure one image is above the rest in the stack.

Most of the styling goes towards the previous and next anchors, which we'll create with the plugin. These are set so that each one will take up exactly half of the container and are positioned to appear side-by-side. We set the `z-index` of these links so that they appear above all of the images. The `font-size` is ramped up considerably, and an excessive `line-height` means we don't need to middle-align the text with `padding`.

In most browsers, we simply set the `color` of the anchors to `transparent`, which hides them. Then we set the `color` to white in the `hover` state. This won't work too well in IE however, so instead we set the link initially to transparent with the Microsoft `opacity filter` and then set it to fully opaque in the `hover`, which serves the same purpose.

Another IE-specific fix

IE also presents us with another problem in that the clickable area of our links will only extend the height of the text within them because of their absolute positioning. We can overcome this by setting a reference to a background-image.

The best part is that the image doesn't even need to exist for the fix to work (so you'll find no corresponding `transparent.gif` file in the book's companion code bundle). The fix has no detrimental effects on normal browsers.

Creating the plugin

Now let's create the plugin itself. Unlike most of the other example code we've looked at, the code for our plugin will go into its own separate file.

Time for action – adding a license and defining configurable options

In a new file create the following outer structure for the plugin:

```
/*
  Plugin name jQuery plugin version 1.0

  Copyright (c) date copyright holder

  License(s)

*/

;(function($) {

  $.tranzify = {

    defaults: {
      transitionWidth: 40,
      transitionHeight: "100%",
      containerID: "overlay",
      transitionType: "venetian",
      prevID: "prev",
      nextID: "next",
      visibleClass: "visible"
    }
  };

})(jQuery);
```

What just happened?

All plugins should contain information on the plugin name and version number, the copyright owner (usually the author of the code) and the terms, or links to the terms, of the license or licenses it is released under.

The plugin is encapsulated within an anonymous function so that its variables are protected from other code which may be in use on the page it is deployed on, and has a semicolon placed before it to ensure it remains a discrete block of code after potential minification, and in case it is used with other, less scrupulously written code than our own.

We also alias the `$` character for safe use within our function, to ensure it is not hijacked by any other libraries running on the page and to preserve the functionality of jQuery's `noConflict()` method.

It is good practice to make plugins as configurable as possible so that end users can adjust it to suit their own requirements. To facilitate this, we should provide a set of default values for any configurable options. When deciding what to make configurable, a good rule of thumb is to hardcode nothing other than pure logic into the plugin. Hence, IDs, class names, anything like that, should be made configurable.

The defaults we set for the plugin are stored in an object that is itself stored as a property of the `jQuery` object that is passed into the function. The property added to the `jQuery` object is called `tranzify`, the name of our plugin, and will be used to store the properties, functions, and methods we create so that all of our code is within a single namespace.

Our default properties are contained in a separate object called `defaults` within the `tranzify` object. We set the `width` and `height` of the transition elements, the `id` of the container that gets created, the default transition, the `id`s for the previous and next links, and the class name we give to the currently-showing image.

As I mentioned, it's best not to hardcode any `id` values or class names into a plugin if possible. The person implementing the plugin may already have an element on the page with an `id` of `overlay` for example, so we should give them the option to change it if need be.

Time for action – adding our plugin method to the jQuery namespace

Next we can add the code that will insert our plugin into the jQuery namespace so that it can be called like other jQuery methods:

```
$.fn.extend({
  tranzify: function(userConfig) {

    var config = (userConfig) ? $.extend({}, $.tranzify.defaults,
      userConfig) : $.tranzify.defaults;

    config.selector = "#" + this.attr("id");

    config.multi = parseInt(this.width()) / config.transitionWidth;

    $.tranzify.createUI(config);

    return this;
  }
});
```

What just happened?

jQuery provides the `fn.extend()` method specifically for adding new methods into jQuery, which is how most plugins are created. We define a function as the value of the sole property of an object passed to the `extend()` method. We also specify that the method may take one argument, which may be a configuration object passed into the method by whoever is using the plugin to change the default properties we have set.

The first thing our method does is check whether or not a configuration object has been passed into the method. If it has, we use the `extend()` method (not `fn.extend()` however) to merge the user's configuration object with our own `defaults` object.

The resulting object, created by the merging of these two objects, is stored in the variable `config` for easy access by our functions. Any properties that are in the `userConfig` object will overwrite the properties stored in our `defaults` object. Properties found in the `defaults` object but not the `userConfig` object will be preserved. If no `userConfig` object is passed into the method, we simply assign the `defaults` object to the `config` variable.

Next we build an `id` selector that matches the element that the method was called on and add this as an extra property to the `config` object, making it convenient to use throughout the plugin. We can't store this as a default property because it is likely to be different on every page that the plugin is used on, and we also can't expect users of the plugin to have to define this in a configuration object each time the plugin is used.

The number of transition elements we need to create will depend on the size of the images, and the width of the transition elements (defined as a configurable property), so we work out a quick multiplier based on the width of the image and the configured transition width for use later on.

Following this we call the function that will create the prev/next links (we define this shortly) and pass the function the `config` object so that it can read any properties that the user has configured.

Finally, we return the jQuery object (which is automatically assigned to the value of the `this` keyword within our plugin method). This is to preserve chaining so that the user can call additional jQuery methods after calling our plugin.

Time for action – creating the UI

Next we need to create the previous and next links that are overlaid above the images and allow the visitor to cycle through the images:

```
$.tranzify.createUI = function(config) {
  var imgLength = $(config.selector).find("img").length,
    prevA = $("<a></a>", {
    id: config.prevID,
    href: "#",
    html: "&laquo;",
    click: function(e) {
      e.preventDefault();

      $(config.selector).find("a").css("display", "none");

      $.tranzify.createOverlay(config);

      var currImg = $("." + config.visibleClass, $(config.selector));
      if(currImg.prev().filter("img").length > 0) {
        currImg.removeClass(config.visibleClass).prev().addClass
         (config.visibleClass);
      } else {
        currImg.removeClass(config.visibleClass);
        $(config.selector).find("img").eq(imgLength -
          1).addClass(config.visibleClass);
      }

      $.tranzify.runTransition(config);

    }
  }).appendTo(config.selector),

  nextA = $("<a></a>", {
    id: config.nextID,
    href: "#",
    html: "&raquo;",
    click: function(e) {
      e.preventDefault();
```

```
        $(config.selector).find("a").css("display", "none");

        $.tranzify.createOverlay(config);

        var currImg = $("." + config.visibleClass, $(config.selector));

        if(currImg.next().filter("img").length > 0) {
          currImg.removeClass(config.visibleClass).next().addClass(
            config.visibleClass);
        } else {
          currImg.removeClass(config.visibleClass);
          $(config.selector).find("img").eq(0).addClass(
            config.visibleClass);
        }

        $.tranzify.runTransition(config);
      }
    }).appendTo(config.selector);
  };
```

What just happened?

This is by far our largest function and deals with creating the previous and next links, as well as defining their click handlers during creation using the jQuery 1.4 syntax. The first thing we do is obtain the number of images in the container as the click handlers we add will need to know this.

We create the anchor for the previous link and in the object passed as the second argument we define the `id` (using the value from the `config` object), a dummy `href`, an HTML entity as its `innerHTML`, and a click handler.

Within the click handler, we use the `preventDefault()` method to stop the browser following the link, then hide the previous and next links in order to protect the widget against multiple clicks, as this will break the transitions.

Next we call our `createOverlay()` function, passing it the `config` object, to create the overlay container and the transition elements. We also cache a reference to the currently selected image using the class name stored in the `config` object.

We then test whether there is another image element before the visible image. If there is, we remove the class from the element that currently has it and give it to the previous image in order to bring it to the top of the stack. If there aren't any more images before the current image, we remove the `visible` class from the current image and move to the last image in the container to show that instead.

Once we've defined everything we need, we can append the new anchor to the specified container. We also create the next link within the current function as well, giving it a very similar set of attributes and a click handler too. All that differs in this click handler is that we test for an image after the current one, and move to the first image in the container if there isn't one.

Time for action – creating the transition overlay

Our next function will deal with creating the overlay and transition elements:

```
$.tranzify.createOverlay = function(config) {

  var posLeftMarker = 0,
    bgHorizMarker = 0

  overlay = $("<div></div>", {
    id: config.containerID
  });

  for (var x = 0; x < multiX; x++) {

    $("<div></div>", {
      width: config.transitionWidth,
      height: config.transitionHeight,
      css: {
        backgroundImage: "url(" + $("." + config.visibleClass,
          $(config.selector)).attr("src") + ")",
        backgroundPosition: bgHorizMarker + "px 0",
        left: posLeftMarker,
        top: 0
      }
    }).appendTo(overlay);
      bgHorizMarker -=config.transitionWidth;
    posLeftMarker +=config.transitionWidth;

  }
  overlay.insertBefore("#" + config.prevID);
};
```

What just happened?

Our next function deals with creating the overlay container and the transition elements that will provide the transition animations. The plugin will need to set the `position` and `background-position` of each transition element differently in order to stack the elements up horizontally. We'll need a couple of counter variables to do this, so we initialize them at the start of the function.

We then create the overlay container `<div>` and give it just an `id` attribute so that we can easily select it when we run the transitions.

Next we create the transition elements. To do this, we use a standard JavaScript `for` loop, which is executed a number of times depending on the multiplier we set earlier in the script. On each iteration of the loop, we create a new `<div>` which has its `width` and `height` set according to the properties stored in the configuration object.

We use the `css()` method to set the `backgroundImage` of the overlay to the currently visible image, and the `backgroundPosition` according to the current value of the `bgHorizMarker` counter variable. We also set the `left` property to position the new element correctly according to the `posLeftMarker` variable, and the `top` property to `0` to ensure correct positioning.

Once created, we append the new element to the container and increment our counter variables. Once the loop exits and we have created and appended all of the transition elements to the container, we can then append the container to the element on the page that the method was called on.

Time for action – defining the transitions

The final function will perform the actual transitions:

```
$.tranzify.runTransition = function(config) {
  var transOverlay = $("#" + config.containerID),
    transEls = transOverlay.children(),
    len = transEls.length - 1;

    switch(config.transitionType) {
      case "venetian":
      transEls.each(function(i) {
        transEls.eq(i).animate({
          width: 0
        }, "slow", function() {

          if (i === len) {
            transOverlay.remove();
```

```
              $(config.selector).find("a").css("display", "block");
            }
          });
        });
        break;
        case "strip":
        var counter = 0;

      function strip() {
        transEls.eq(counter).animate({
          height: 0
        }, 150, function() {

          if (counter === len) {
            transOverlay.remove();
            $(config.selector).find("a").css("display", "block");
          } else {
            counter++;
            strip();
          }
        });
      }
      strip();
    }
  };
```

What just happened?

Our last function deals with actually running the transitions. In this example, there are just two different types of transitions, but we could easily extend this to add more transition effects.

This function also requires some variables, so we set these at the start of the function for later use. We cache a reference to the overlay container as we'll be referring to it several times. We also store the collection of transition elements, and the number of transition elements. We subtract `1` from the number of children because the figure will be used with jQuery's `eq()` method, which is zero-based.

To determine which of our transitions to run, we use a JavaScript `switch` statement and check the value of the `config.transitionType` property. The first transition is a kind of venetian-blind effect. To run this transition, we just animate the `width` of each element to `0` using the jQuery `each()` method. The function we specify as the argument to this method automatically receives the index of the current element, which we access using `i`.

In the callback function for each animation, we check whether `i` is equal to the `length` of the collection of transition elements, and if it is we remove the overlay and show the previous and next links once more.

The second transition removes the old image one strip at a time. To do this, we use a simple `counter` variable and a standard JavaScript function. We can't use the `each()` method this time, or all of the transition elements will slide down together, but we want each one to slide down on its own.

Within the function, we animate the current transition element's height to `0` and set a rather low duration so that it happens fairly quickly. If the animation is too slow it spoils the effect. In the callback function, we check whether our `counter` variable is equal to the number of transition elements, and if so remove the overlay and show the links again. If the `counter` hasn't reached the last element at this point, we increment the `counter` variable and call the function once more.

Save this file as `jquery.tranzify.js` in the `js` folder. This is the standard naming convention for jQuery plugins and should be adhered to.

Using the plugin

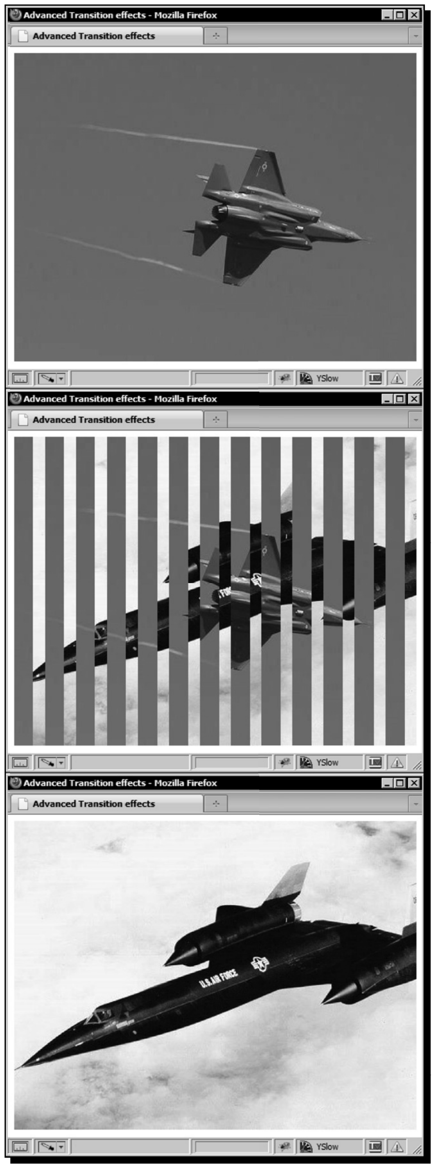
To use the plugin, we just call it like we would call any other jQuery method, like this:

```
$("#frame").tranzify();
```

In this form, the default properties will be used. If we wanted to change one of the properties, we just supply a configuration object, such as this:

```
$("#frame").tranzify({
  transitionType: "strip"
});
```

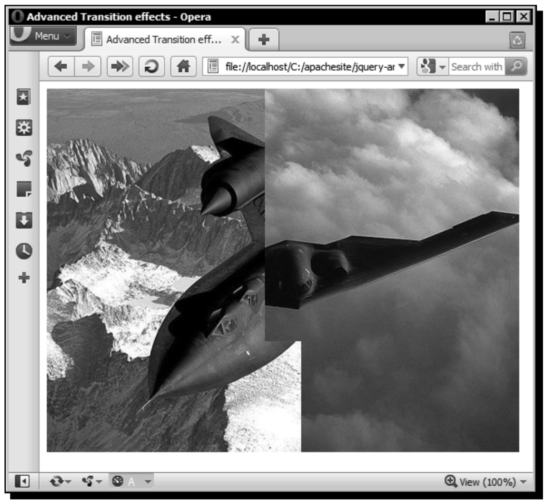
The default animation should run something like this:


Advanced Transition effects - Mozilla Firefox
Advanced Transition effects
YSlow

In the previous screenshot, we see the transition elements all simultaneously shrinking to `0 width`, creating an effect like Venetian blinds being opened to reveal the new image.

Using the plugin is simple; there is just one point to remember. The images should all be the same size, and the `width` of each image should be exactly divisible by the `transitionWidth` property. As we've exposed the `transitionWidth` as a configurable property, we should be able to use any size image we wish and set this accordingly.

For reference, the second transition effect runs like this, with strips of the old image sliding away to reveal the new image:


Advanced Transition effects - Opera
Menu
Advanced Transition eff...
file://localhost/C:/apachesite/jquery-ar
Search with
View (100%)

In the previous screenshot, we can see the effects of the second transition type, with the old image being stripped away to reveal the new image.

Pop quiz – creating a plugin

1. What is the difference between a plugin method and a function?
   a. There is no difference, conceptually and in practice they are the same
   b. Methods are able to accept arguments, functions are not
   c. Methods execute faster
   d. Methods are attached to the `fn` object and are used like existing jQuery methods, while functions are attached directly to the jQuery object and called like any normal function
2. What must each new method return?
   a. A string containing the `id` attribute of the selected element
   b. An array containing the `id` attributes of selected elements
   c. The `this` object, which points to the currently selected element
   d. Nothing should be returned

Have a go hero – extending the plugin

Our plugin currently contains just two transition effects (venetian and strip). Extend the plugin to include more transition effects of your own devising. The plugin currently creates a number of transition elements that are the full height of each image.

By wrapping our existing `for` loop within another `for` loop and adding some new counter variables for `top` position and vertical `background-position`, it is relatively easy to add square transition elements in a checker-board style, which opens up the possibility of more complex, and attractive, transition effects. Do this.

Summary

In this chapter, we looked at some common usages of the `animate()` method, which is the means for us to create custom animations in jQuery when the built-in effects are not enough for our requirements. The method is robust, easy to use, and makes complex animations trivial.

When simple sliding or fading does not meet our requirements, we can fall back onto the `animate()` method in order to craft our own high-quality custom animations. We learnt the following points about the method:

- The `animate()` method can be used to animate any numeric CSS property (except colors, for which a separate plugin is required).
- The arguments passed into the method may take one of two formats. The first allows us to pass in an object containing the CSS properties to animate, as well as separate duration, easing, and callback arguments. The second format allows us to pass in two objects, the first allowing us to specify the CSS properties to animate as before, and the second allowing us to specify additional options such as the duration, easing, and callback. The second option gives us access to some special arguments not accessible in the first format such as `specialEasing` and the `step` callback.
- All CSS properties specified in the first object will be executed simultaneously.
- How to achieve animations involving an element's position, or its dimensions

We also looked at how we can extend the jQuery library with brand new functions and methods in the form of plugins. Plugins are a great way of wrapping up code for easy deployment and sharing.

Now that we've looked at all of jQuery's animation methods, we're going to move on and take a look at the additional animation functionality provided by the excellent jQuery UI library. The next chapter will cover all of the additional effects added by the UI library, as well as look at class transitioning and smooth color animating.