CHAPTER

8

Basic Drawing with the Canvas

As you learned in Chapter 1, one of HTML5's goals is to make it easier to put *rich applications* inside otherwise ordinary web pages. In this case, the word "rich" doesn't have anything to do with your bank account. Instead, a rich application is one that's decked out with slick graphics, interactive features, and showy frills like animation.

One of the most important HTML5 tools for rich applications is the *canvas*, a drawing surface where you can let your inner Picasso loose. Compared with every other HTML element, the canvas is unique because it *requires* JavaScript. There's no way to draw shapes or paint pictures without it. That means the canvas is essentially a programming tool—one that takes you far beyond the original document-based idea of the Web.

At first, using the canvas can feel like stuffing your page with a crude version of Windows Paint. But dig deeper, and you'll discover that the canvas is the key to a range of graphically advanced applications, including some you've probably already thought about (like games, mapping tools, and dynamic charts) and others that you might not have imagined (like musical lightshows and physics simulators). In the not-so-distant past, these applications were extremely difficult without the help of a browser plug-in like Flash. Today, with the canvas, they're all possible, provided you're willing to put in a fair bit of work.

In this chapter, you'll learn how to create a canvas and fill it up with lines, curves, and simple shapes. Then you'll put your skills to use by building a simple painting program. And, perhaps most importantly, you'll learn how you can get canvas-equipped pages to work on old browsers that don't support HTML5.

NOTE For some developers, the canvas will be indispensable. For others, it will just be an interesting diversion. (And for some, it may be interesting but still way too much work compared with a mature programming platform like Flash.) But one thing is certain: This straightforward drawing surface is destined to be much more than a toy for bored programmers.

Getting Started with the Canvas

The <canvas> element is the place where all your drawing takes place. From a markup point of view, it's as simple as can be. You supply three attributes: `id`, `width`, and `height`:

```
<canvas id="drawingCanvas" width="500" height="300"></canvas>
```

The `id` attribute gives the canvas a unique name, which you'll need when your script code goes searching for it. The `width` and `height` attributes set the size of your canvas, in pixels.

NOTE You should always set the size of your canvas through the width and height attributes, not the width and height style sheet properties. To learn about the possible problem that can occur if you use style sheet sizing, see the box on page 277.

Ordinarily, the canvas shows up as a blank, borderless rectangle (which is to say it doesn't show up at all). To make it stand out on the page, you can apply a background color or a border with a style sheet rule like this:

```
canvas {
  border: 1px dashed black;
}
```

Figure 8-1 shows this starting point.

To work with a canvas, you need to fire off a bit of JavaScript that takes two steps. First, your script must use the indispensable `document.getElementById()` method to grab hold of the canvas object. In this example, you'll name the canvas `drawingCanvas`, so the code looks like this:

```
var canvas = document.getElementById("drawingCanvas");
```

This code is nothing new, as you use the `getElementById()` method whenever you need to find an HTML element on your page.

NOTE If you aren't familiar with JavaScript, you won't get far with the canvas. To brush up with the absolute bare-minimum essentials, read Appendix B, "JavaScript: The Brains of Your Page."

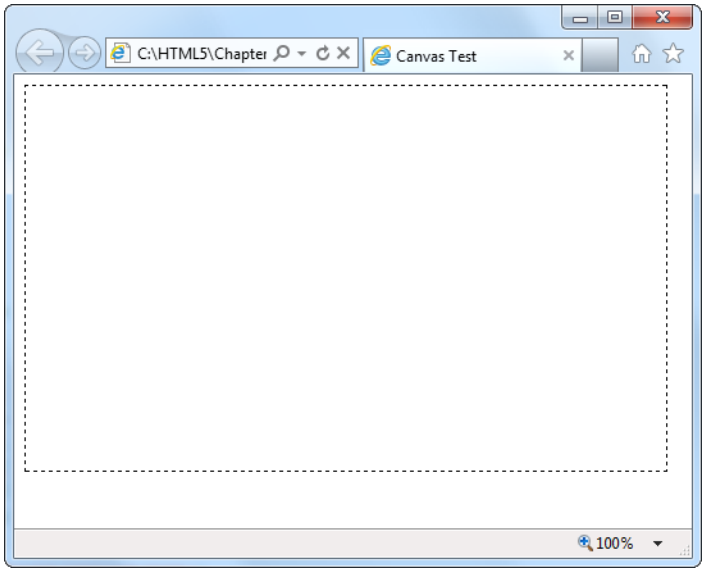

FIGURE 8-1

Every canvas begins as an empty rectangle somewhere on a web page. To put even a single line inside, you need to write some JavaScript code.

Once you have the canvas object, you can take the second essential step. You use the canvas object's `getContext()` method to retrieve a two-dimensional *drawing context*, like this:

```
var context = canvas.getContext("2d");
```

You can think of the context as a supercharged drawing tool that handles all your canvas tasks, like painting rectangles, writing text, pasting an image, and so on. It's a sort of one-stop shop for canvas drawing operations.

NOTE The fact that the context is explicitly called *two-dimensional* (and referred to as `2d` in the code) raises an obvious question—namely, is there a three-dimensional drawing context? Not yet, but the creators of HTML5 have clearly left space for one in the future.

You can grab the context object and start drawing at any point: for example, when the page first loads, when the visitor clicks a button, or at some other point. When you're just starting out with the canvas, you probably want to create a practice page that gets to work straightaway. Here's a template for a page that does just that:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Canvas Test</title>
```

```
  <style>
canvas {
  border: 1px dashed black;
}
  </style>

  <script>
window.onload = function() {
  var canvas = document.getElementById("drawingCanvas");
  var context = canvas.getContext("2d");

  // (Put your fabulous drawing code here.)
};
  </script>
</head>

<body>
  <canvas id="drawingCanvas" width="500" height="300"></canvas>
</body>
</html>
```

The <body> of this page includes the <canvas> element and no other markup. The <style> section of this page makes the canvas stand out with a border. The <script> section handles the window.onload event, which occurs once the browser has completely loaded the page. The code then gets the canvas, creates a drawing context, and gets ready to draw. You can use this example as the starting point for your own canvas experiments.

NOTE Of course, when you're using the canvas in a real page on your website, you'll want to declutter a bit by snipping out the JavaScript code and putting it in an external script file (page 455). But for now, this template gives you single-page convenience. If you want to type the examples in on your own, you can get this markup from the CanvasTemplate.html file on the try-out site (*http://prosetech.com/html5*).

Straight Lines

Now you're just about ready to start drawing. But before you add anything on a canvas, you need to understand its coordinate system. Figure 8-2 shows you how it works.

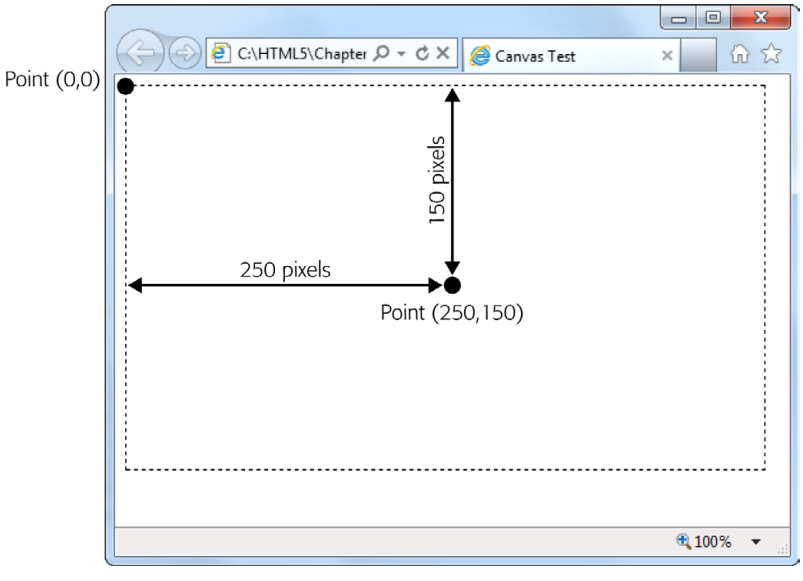

FIGURE 8-2

As with every other HTML element, the canvas designates its top-left corner as the point (0,0). As you move right, the x-value increases, and as you move down, the y-value increases. If a canvas is 500 x 300 pixels, then the bottom-right corner is point (500, 300).

The simplest thing you can draw on a canvas is a solid line. Doing that takes three actions with the drawing context. First, you use the `moveTo()` method to move to the point where you want the line to start. Second, you use the `lineTo()` method to travel from the current point to the end of the line. Third, you call the `stroke()` method to make the line actually appear:

```
context.moveTo(10,10);
context.lineTo(400,40);
context.stroke();
```

Or think of it this way: First you lift up your pen and put it where you want (using `moveTo`), then you drag the pen across the canvas (using `lineTo`), then you make the line appear (using `stroke`). This result is a thin (1-pixel) black line from point (10,10) to point (400,40).

Happily, you can get a little fancier with your lines. At any point before you call the `stroke()` method that winks your line into existence, you can set three drawing context properties: `lineWidth`, `strokeStyle`, and `lineCap`. These properties affect everything you draw from that point on, until you change them.

You use `lineWidth` to set the width of your lines, in pixels. Here's a thick, 10-pixel line:

```
context.lineWidth = 10;
```

You use strokeStyle to set the color of your lines. You can use an HTML color name, an HTML color code, or the CSS rgb() function which lets you assemble a color from red, green, and blue components. (This approach is useful because most drawing and painting programs use the RGB system.) No matter which one you use, you need to wrap the whole value in quotation marks, as shown here:

```
// Set the color (brick red) using an HTML color code:
context.strokeStyle = "#cd2828";

// Set the color (brick red) using the rgb() function:
context.strokeStyle = "rgb(205,40,40)";
```

NOTE This property is named strokeStyle rather than strokeColor because you aren't limited to plain colors. As you'll see later on, you can use color blends called gradients (page 284) and image-based patterns (page 283).

Finally, use lineCap to decide how you want to cap off the ends of your lines. The default is to make a squared-off edge with butt, but you can also use round (to round off the edge) or square (which looks the same as butt, but extends the line an amount equal to half its thickness on each end).

And here's the complete script code you need to draw three horizontal lines, with different line caps (Figure 8-3). To try this code out, pop it into any JavaScript function you want. To make it run right away, put it in the function that handles the window.onload event, as shown on page 248:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Set the line width and color (for all the lines).
context.lineWidth = 20;
context.strokeStyle = "rgb(205,40,40)";

// Draw the first line, with the standard butt ending.
context.moveTo(10,50);
context.lineTo(400,50);
context.lineCap = "butt";
context.stroke();

// Draw the second line, with a round cap.
context.beginPath();
context.moveTo(10,120);
context.lineTo(400,120);
context.lineCap = "round";
context.stroke();
```

```
// Draw the third line, with a square cap.
context.beginPath();
context.moveTo(10,190);
context.lineTo(400,190);
context.lineCap = "square";
context.stroke();
```

This example introduces one new feature: the `beginPath()` method of the drawing context. When you call `beginPath()`, you start a new, separate segment of your drawing. Without this step, every time you call `stroke()`, the canvas will attempt to draw everything over again. This is a particular problem if you're changing other context properties. In this case, you'd end up drawing over your existing content with the same shapes but a new color, thickness, or line cap.

NOTE While you do need to begin new segments by calling `beginPath()`, you don't need to do anything special to end a segment. Instead, the current segment is automatically considered "finished" the moment you create a new segment.

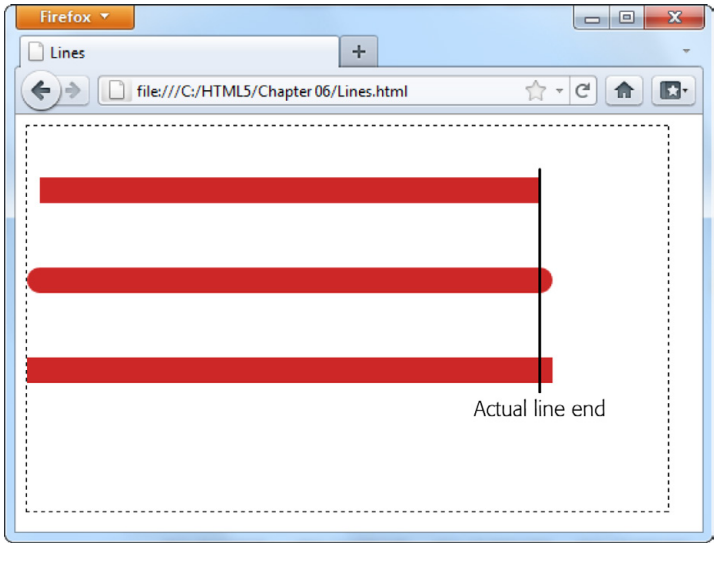

FIGURE 8-3

The top line uses the standard `butt` *ending, while the lines below have added caps (*`round` *or* `square`*), which extend the line by an amount equal to half the line's thickness.*

Paths and Shapes

In the previous example, you separated different lines by starting a new path for each one. This method lets you give each line a different color (and a different width and cap style). Paths are also important because they allow you to fill custom shapes. For example, imagine you create a red-outlined triangle using this code:

```
context.moveTo(250,50);
context.lineTo(50,250);
context.lineTo(450,250);
context.lineTo(250,50);

context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

But if you want to *fill* that triangle, the `stroke()` method won't help you. Instead, you need to close the path by calling `closePath()`, pick a fill color by setting the `fillStyle` property, and then call the `fill()` method to make it happen:

```
context.closePath();
context.fillStyle = "blue";
context.fill();
```

It's worth tweaking a couple of things in this example. First, when closing a path, you don't need to draw the final line segment, because calling `closePath()` automatically draws a line between the last drawn point and the starting point. Second, it's best to fill your shape first, and *then* draw its outline. Otherwise, your outline may be partially overwritten by the fill.

Here's the complete triangle-drawing code:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

context.moveTo(250,50);
context.lineTo(50,250);
context.lineTo(450,250);
context.closePath();

// Paint the inside.
context.fillStyle = "blue";
context.fill();

// Draw the outline.
context.lineWidth = 10;
context.strokeStyle = "red";
context.stroke();
```

Notice that you don't need to use `beginPath()` in this example, because the canvas starts you off with a new path automatically. You need to call `beginPath()` only when you need a *new* path—for example, when changing line settings or drawing a new shape. Figure 8-4 shows the result of running this JavaScript.

NOTE When drawing connecting line segments (like the three sides of this triangle), you can set the drawing context's `lineJoin` property to round or bevel the edges (by using the values `round` or `bevel`—the default is `mitre`).

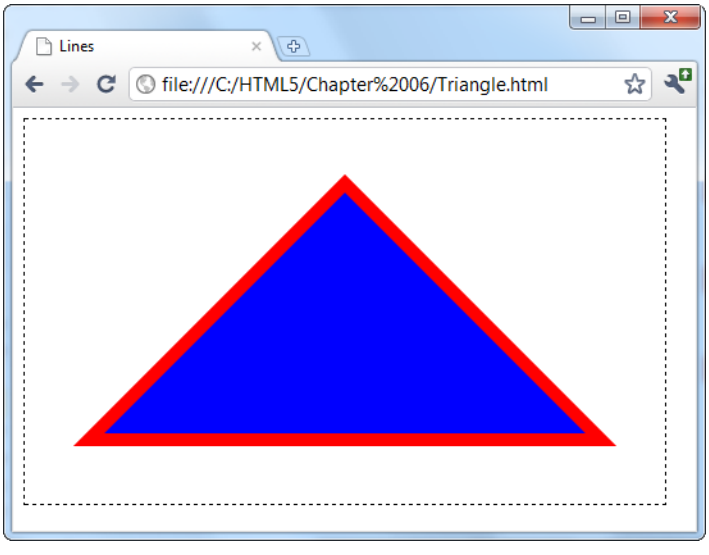

FIGURE 8-4
To create a closed shape like this triangle, use `moveTo()` *to get to the starting point,* `lineTo()` *to draw each line segment, and* `closePath()` *to complete the shape. You can then fill it with* `fill()` *and outline it with* `stroke()`.

Most of the time, when you want a complex shape, you'll need to assemble a path for it, one line at a time. But there's one shape that's important enough to get special treatment: the rectangle. You can fill a rectangular region in one step using the `fillRect()` method. You supply the coordinate for the top-left corner, the width, and the height.

For example, to place a 100 x 200 pixel rectangle starting at point (0,10), use this code:

```
fillRect(0,10,100,200);
```

The `fillRect()` method gets the color to use from the `fillStyle` property, just like the `fill()` method.

Similarly, you can use `strokeRect()` to draw the outline of a rectangle in one step:

```
strokeRect(0,10,100,200);
```

The `strokeRect()` method uses the current `lineWidth` and `strokeStyle` properties to determine the thickness and color of the outline, just as the `stroke()` method does.

Curved Lines

If you want something more impressive than lines and rectangles (and who doesn't?), you'll need to understand four methods that can really throw you for a curve: `arc()`, `arcTo()`, `bezierCurveTo()`, and `quadraticCurveTo()`. All of these methods draw curved lines in different ways, and they all require at least a smattering of math (and some need a whole lot more).

The `arc()` method is the simplest of the bunch. It draws a portion of a circle's outline. To draw an arc, you first need to visualize an imaginary circle, and then decide which part of the edge you need, as explained in Figure 8-5. You'll then have all the data you need to pass to the `arc()` method.

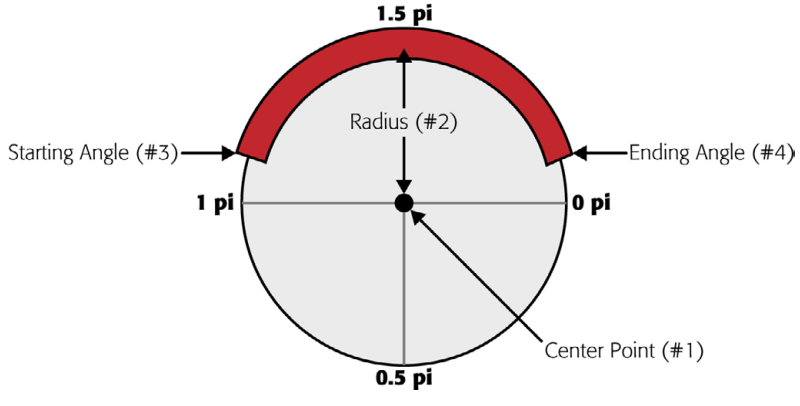

FIGURE 8-5

An arc seems simple enough, but you need several pieces of information to describe it completely. First, you need to pin down the imaginary circle. That requires the coordinates of the center point (#1) and the radius that indicates how big the circle is (#2). Next, you need to describe the length of the arc, which requires the angle where the arc starts (#3) and the angle where it ends (#4). You must supply angles in radian coordinates, which are expressed as fractions of the constant pi. (So 1 pi is halfway around the circle, and 2 pi is all the way around, as indicated here.)

Once you've sorted out all the details you need, you can call the `arc()` method:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Create variables to store each detail about the arc.
var centerX = 150;
var centerY = 300;
var radius = 100;
var startingAngle = 1.25 * Math.PI;
var endingAngle = 1.75 * Math.PI;
```

```
// Use this information to draw the arc.
context.arc(centerX, centerY, radius, startingAngle, endingAngle);
context.stroke();
```

Or, call `closePath()` before you call `stroke()` to add a straight line between the two ends of the arc. This creates a closed semi-circle.

Incidentally, a circle is simply an arc that stretches all the way around. You can draw it like this:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

var centerX = 150;
var centerY = 300;
var radius = 100;
var startingAngle = 0;
var endingAngle = 2 * Math.PI;

context.arc(centerX, centerY, radius, startingAngle, endingAngle);
context.stroke();
```

NOTE The `arc()` method doesn't let you draw an ellipse (a flattened circle). To get that, you need to do more work—either use some of the more sophisticated curve methods described next, or use a transform (page 256) to stretch out an ordinary circle as you draw it.

The three other curve methods—`arcTo()`, `bezierCurveTo()`, and `quadraticCurveTo()`—are a bit more intimidating to the geometrically challenged. They involve a concept called *control points*—points that aren't included in the curve, but influence the way it's drawn. The most famous example is the Bézier curve, which is used in virtually every computer illustration program ever created. It's popular because it creates a curve that looks smooth no matter how small or big you draw it. Figure 8-6 shows how control points shape a Bézier curve.

And here's the code that creates the curve from Figure 8-6:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Put the pen where the curve starts.
context.moveTo(62, 242);

// Create variables for the two control points and the end point of the curve.
var control1_x = 187;
var control1_y = 32;
var control2_x = 429;
var control2_y = 480;
```

```
var endPointX = 365;
var endPointY = 133;

// Draw the curve.
context.bezierCurveTo(control1_x, control1_y, control2_x, control2_y,
 endPointX, endPointY);
context.stroke();
```

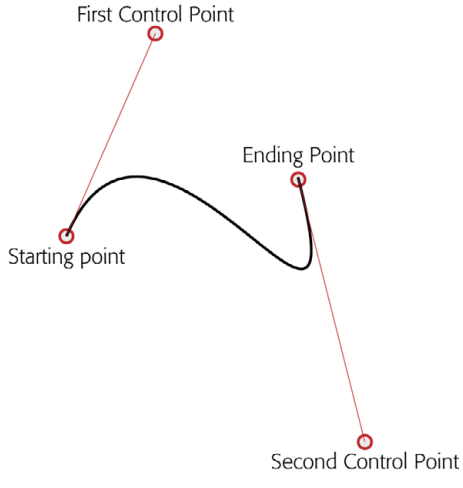

FIGURE 8-6

A Bézier curve has two control points. The start of the curve runs parallel to the first control point. The end of the curve runs parallel to the second control point. In between, the line curves. The amount of curvature is influenced by the distance to the control point—the farther away the point is, the stronger its "pull." It's sort of like gravity, but in reverse.

The outline of a complex, organic shape often involves a series of arcs and curves glued together. Once you're finished, you can call `closePath()` to fill or outline the entire shape. The best way to learn about curves is to play with one on your own. You can find a perfect test page at *http://tinyurl.com/html5bezier* (Figure 8-7).

Transforms

A transform is a drawing technique that lets you shift the canvas's coordinate system. For example, imagine you want to draw the same square in three places. You could call `fillRect()` three times, with three different points:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Draw a 30x30 square, at three places.
context.rect(0, 0, 30, 30);
context.rect(50, 50, 30, 30);
context.rect(100, 100, 30, 30);

context.stroke();
```

Or you could call `fillRect()` three times, with the *same* point, but shift the coordinate system each time so the square actually ends up in three different spots, like so:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Draw a square at (0,0).
context.rect(0, 0, 30, 30);

// Shift the coordinate system down 50 pixels and right 50 pixels.
context.translate(50, 50);
context.rect(0, 0, 30, 30);

// Shift the coordinate system down a bit more. Transforms are cumulative,
// so now the (0,0) point will actually be at (100,100).
context.translate(50, 50);
context.rect(0, 0, 30, 30);

context.stroke();
```

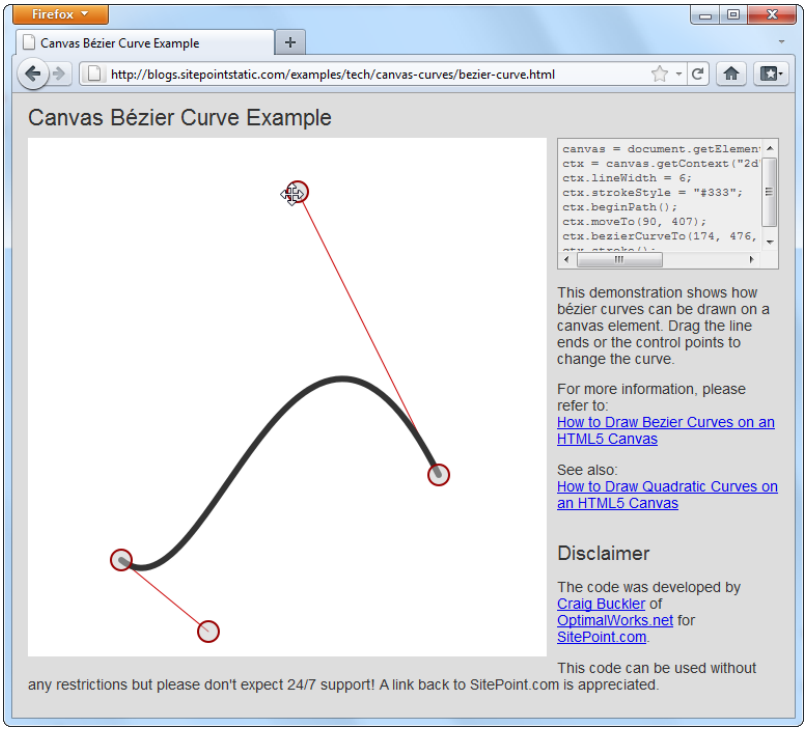

FIGURE 8-7

This page (found at http://tinyurl.com/html5bezier) lets you tweak all the details of a Bézier curve by clicking and pulling with the mouse. Best of all, as you drag the starting point, control points, and end point, the page generates the corresponding snippet of HTML5 canvas code that you can use to create the same curve on your own canvas. You can find a similarly great test page for quadratic curves at http://tinyurl.com/html5quadratic.

FREQUENTLY ASKED QUESTION

Canvas Drawing for Math-Phobes

How do I get all the shapes with none of the headaches?

If you're hoping to use the canvas to create eye-catching graphics, but you don't want to pick up a degree in geometry, you might be a bit frustrated. Fortunately, there are several approaches that can help you draw what you want without worrying about the mathematical underpinnings:

- **Use a drawing library.** Why draw everything the hard way when you can use someone else's drawing library to draw circles, triangles, ellipses, and polygons in a single step? The idea is simple—you call a higher-level method (say, `fillEllipse()`, with the appropriate coordinates), and the JavaScript library translates that to the correct canvas operations. Two good examples are Fabric.js (*http://fabricjs.com*) and KineticJS (*http://kineticjs.com*). However, these libraries (and more) are still evolving—and rapidly. It's too soon to say which ones will have real staying power, but you could read a lively debate and some developer suggestion on the popular question-and-answer Stack Overflow site (*http://tinyurl.com/canvas-libraries*).
- **Draw bitmap images.** Instead of painstakingly drawing each shape you need, you can copy ready-made graphics to your canvas. For example, if you have an image of a circle with a file name *circle.png*, you can insert that into your canvas using the approach shown on page 276. However, this technique won't give you the same flexibility to manipulate your image (for example, to stretch it, rearrange it, remove part of it, and so on).
- **Use an export tool.** If you have a complex graphic and you need to manipulate it on the canvas or make it interactive, drawing a fixed bitmap isn't good enough. But a conversion tool that can examine your graphic and generate the right canvas-creation code just might solve your problem. One intriguing example is the Ai⟶Canvas plug-in for Adobe Illustrator (*http://visitmix.com/labs/ai2canvas*), which converts Adobe Illustrator artwork to an HTML page with JavaScript code that painstakingly recreates the picture on a canvas.

Both versions of this code have the same effect: They draw three squares, in the same three spots.

At first glance, transforms may seem like nothing more than a way to make a somewhat complicated drawing task even more complicated. But transforms can work magic in some tricky situations. For example, suppose you have a function that draws a series of complex shapes that, put together, create a picture of a bird. Now, say you want to animate that bird, so it appears to fly around the canvas. (You'll see a basic example of animation on the canvas on page 301.)

Without transforms, you'd need to adjust every coordinate in your drawing code each time you drew the bird. But with transforms, you can leave your drawing code untouched and simply tweak the coordinate system over and over again.

Transforms come in several different flavors. In the previous example, a `translate` transform was used to move the center point of the coordinate system—that's the (0,0) point that's usually placed in the top-left corner of the canvas. Along with the translate transform, there's also a `scale` transform (which lets you draw things

bigger or smaller), a `rotate` transform (which lets you turn the coordinate system around), and a `matrix` transform (which lets you stretch and warp the coordinate system in virtually any way—provided you understand the complex matrix math that underpins the visual effect you want).

Transforms are cumulative. The following example moves the (0,0) point to (100,100) with a translate transform and then rotates the coordinate system around that point several times. Each time, it draws a new square, creating the pattern shown in Figure 8-8:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Move the (0,0) point. This is important, because
// the rotate transform turns around this point.
context.translate(100, 100);

// Draw 10 squares.
var copies = 10;
for (var i=1; i<copies; i++) {
  // Before drawing the square, rotate the coordinate system.
  // A complete rotation is 2*Math.PI. This code does a fraction of this
  // for each square, so that it has rotated around completely by the time
  // it's drawn the last one.
  context.rotate(2 * Math.PI * 1/(copies-1));

  // Draw the square.
  context.rect(0, 0, 60, 60);
}
context.stroke();
```

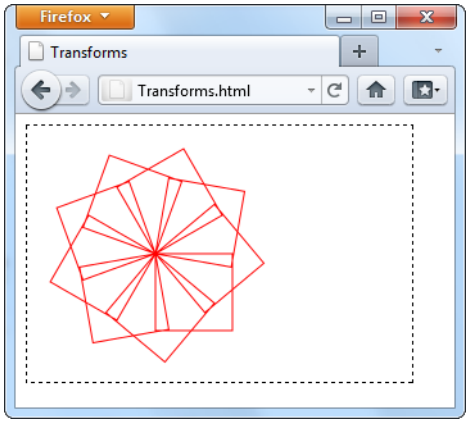

FIGURE 8-8

By drawing a series of rotated squares, you can create Spirograph-like patterns.

TIP You can use the drawing context's `save()` method to save the current state of the coordinate system. Later on, you can use the `restore()` method to return to your previous saved state. You might want to call `save()` before you've applied any transforms, so you can call `restore()` to get the coordinate system back to normal. And in long, complex drawing tasks, you might save the state many times. This list of saved states acts like the web page history in a browser. Each time you call `restore()`, the coordinate system reverts to the immediately preceding state.

Transforms are somewhat beyond the scope of this chapter. If you want to explore them in more detail, Mozilla (the company that created Firefox) has some helpful documentation and examples at *http://tinyurl.com/canvas-transforms*.

Transparency

So far, you've been dealing with solid colors. However, the canvas also lets you use partial transparency to layer one shape over another. There are two ways to use transparency with the canvas. The first approach is to set a color (through the `fillStyle` or `strokeStyle` properties) with the `rgba()` function, instead of the more common `rgb()` function. The `rgba()` function takes four arguments—the numbers for the red, green, and blue color components (from 0 to 255), and an additional number for the alpha value (from 0 to 1), which sets the color's opacity. An alpha value of 1 is completely solid, while an alpha value of 0 is completely invisible. Set a value in between—for example, 0.5—and you get a partially transparent color that any content underneath shows through.

NOTE What content is underneath and what content is on top depends solely on the order of your drawing operations. For example, if you draw a circle first, and then a square at the same location, the square will be superimposed on top of the circle.

Here's an example that draws a circle and a triangle. They both use the same fill color, except that the triangle sets the alpha value to 0.5, making it 50 percent opaque:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Set the fill and outline colors.
context.fillStyle = "rgb(100,150,185)";
context.lineWidth = 10;
context.strokeStyle = "red";

// Draw a circle.
context.arc(110, 120, 100, 0, 2*Math.PI);
context.fill();
context.stroke();
```

```
// Remember to call beginPath() before adding a new shape.
// Otherwise, the outlines of both shapes will
// be merged together in an unpredictable way.
context.beginPath();

// Give the triangle a transparent fill.
context.fillStyle = "rgba(100,150,185,0.5)";

// Now draw the triangle.
context.moveTo(215,50);
context.lineTo(15,250);
context.lineTo(315,250);
context.closePath();
context.fill();
context.stroke();
```

Figure 8-9 shows the result.

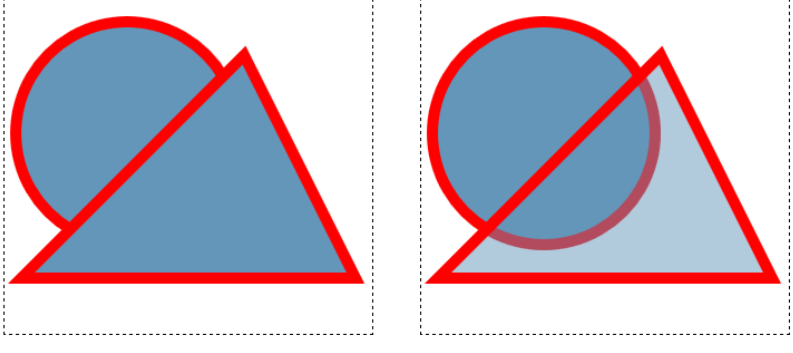

FIGURE 8-9

Left: Two solid shapes, one on top of the other.

Right: One solid shape, with a semitransparent shape on top. Semitransparent shapes look lighter (because they let the white background through), and they allow you to see whatever content you've drawn underneath. Notice that in this example, the semitransparent shape uses a fully opaque color for its border.

The other way to use transparency is to set the drawing context's `globalAlpha` property, like this:

```
context.globalAlpha = 0.5;

// Now this color automatically gets an alpha value of 0.5:
context.fillStyle = "rgb(100,150,185)";
```

Do that, and everything you draw from that point on (until you change `globalAlpha` again) will use the same alpha value and get the same degree of transparency. This includes both stroke colors and fill colors.

So which approach is better? If you need just a single transparent color, use `rgba()`. If you need to paint a variety of shapes with different colors, and they all need the same level of transparency, use `globalAlpha`. The `globalAlpha` property is also useful if you want to paint semitransparent images on your canvas, as you'll learn to do on page 301.

Composite Operations

So far, this chapter has assumed that when you put one shape on top of another, the second shape paints over the first, obscuring it. The canvas works this way most of the time. However, the canvas also has the ability to use more complex *composite operations*.

A composite operation is a rule that tells the canvas how to display two images that overlap. The default composite operation is `source-over`, which tells the canvas that the new shape should be painted *over* the first shape (that is, on top of the first shape). If the new shape overlaps with the first shape, the new shape obscures it.

NOTE In the lingo of composite operations, the *source* is the new object you're drawing, and the *destination* is the existing content on the canvas that you've already drawn.

But other composition options are possible. For example, you can use `xor`, which tells the canvas to show nothing at all in the area where the shapes overlap. Figure 8-10 shows an overview of the different composite operations.

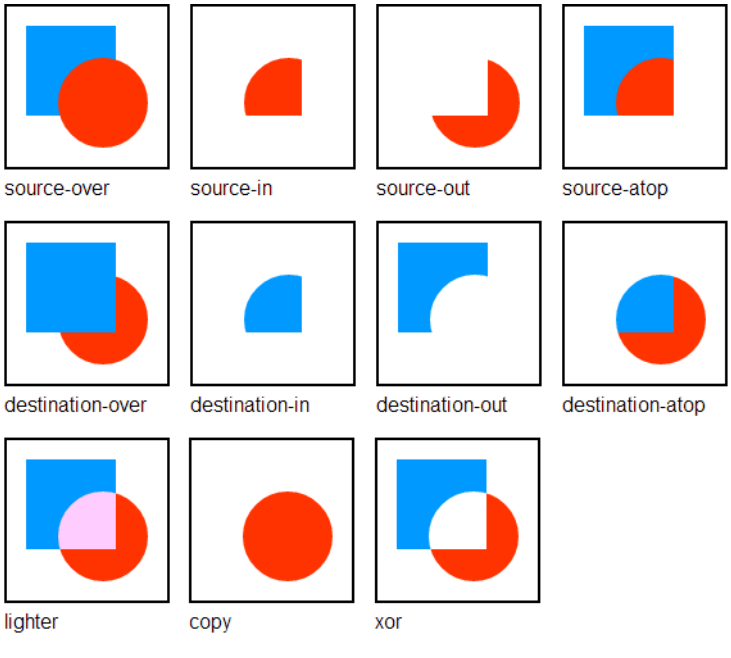

FIGURE 8-10
Here are the 11 possible composite operations and their effects. There used to be 12, but browser makers couldn't agree on how to perform the calculations for the `darker` *composite operation, and it was dropped.*

To change the current composite operation that the canvas uses, simply set the drawing context's `globalCompositeOperation` property before you draw the second shape, like this:

```
context.globalCompositeOperation = "xor";
```

For example, to create the `source-atop` combination shown in the top-right corner of Figure 8-10, you'd use this code:

```
var canvas = document.getElementById("drawingCanvas");
var context = canvas.getContext("2d");

// Draw a rectangle.
context.fillStyle = "blue";
context.fillRect(15,15,70,70);

// Choose the global composite operation.
context.globalCompositeOperation = "source-atop";

// Draw a circle overtop.
context.fillStyle = "red";
context.beginPath();
context.arc(75, 75, 35, 0, Math.PI*2, true);
context.fill();
```

Used cleverly, a composite operation can provide shortcuts for certain drawing tasks, like drawing complex shapes. Hard-core canvas coders can even use these shortcuts to improve performance by reducing the number of drawing operations they perform.

In the recent past, browsers didn't quite agree on how to deal with certain composite operations. Fortunately, these quirks have now been ironed out. The only issue is with any polyfills that you use to get canvas support on old browsers. Right now, the only polyfill that supports composite operations is FlashCanvas Pro (page 273).

Building a Basic Paint Program

The canvas still has a fair bit more in store for you. But you've covered enough ground to build your first practical canvas-powered program. It's the simple painting program shown in Figure 8-11.

The JavaScript that makes this example work is longer than the examples you've seen so far, but still surprisingly straightforward. You'll consider it piece by piece in the following sections.

TIP If you're curious about the style sheet rules that create the blue toolbars above and below the canvas, you want to see the whole example in one piece, or you just want to paint something in your browser, then you can use the *Paint.html* file on the try-out site (*http://prosetech.com/html5*).

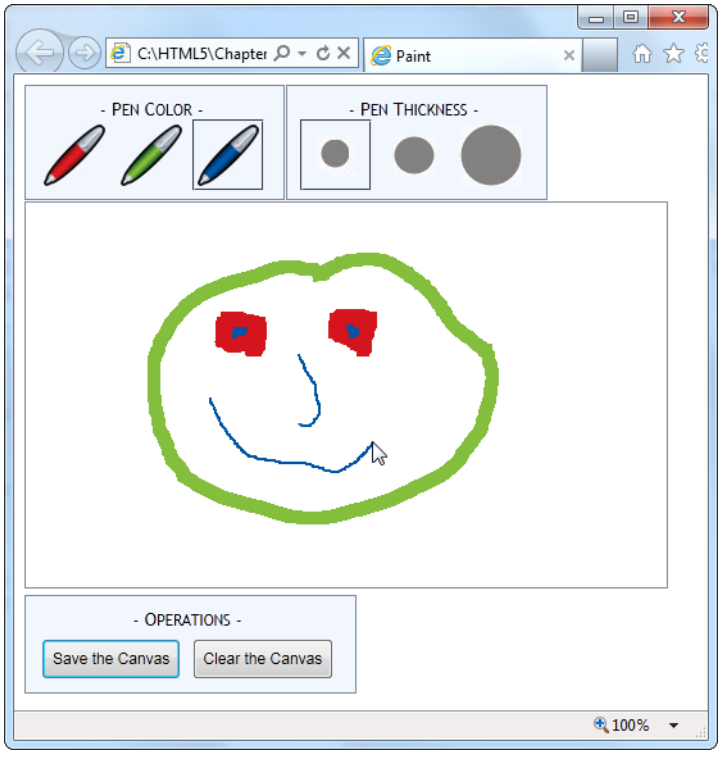

FIGURE 8-11

To express yourself with this paint program, just pick a pen color, pick a pen thickness, and scribble away with the mouse.

Preparing to Draw

First, when the page loads, the code grabs the canvas object and attaches functions that will handle several JavaScript events for different mouse actions: `onMouseDown`, `onMouseUp`, `onMouseOut`, and `onMouseMove`. (As you'll see, these events control the drawing process.) At the same time, the page also stores the canvas in a handy global variable (named `canvas`), and the drawing context in another global variable (named `context`). This way, these objects will be easily available to the rest of the code:

```
var canvas;
var context;

window.onload = function() {
  // Get the canvas and the drawing context.
  canvas = document.getElementById("drawingCanvas");
  context = canvas.getContext("2d");
```

```
  // Attach the events that you need for drawing.
  canvas.onmousedown = startDrawing;
  canvas.onmouseup = stopDrawing;
  canvas.onmouseout = stopDrawing;
  canvas.onmousemove = draw;
};
```

To get started with the paint program, the person using the page chooses the pen color and pen thickness from the two toolbars at the top of the window. These toolbars are simple elements styled to look like nice steel-blue boxes, with a handful of clickable <img> elements in them. For example, here's the toolbar with the three color choices:

```
<div class="Toolbar">
  - Pen Color -<br>
  <img id="redPen" src="pen_red.gif" alt="Red Pen"
   onclick="changeColor('rgb(212,21,29)', this)">
  <img id="greenPen" src="pen_green.gif" alt="Green Pen"
   onclick="changeColor('rgb(131,190,61)', this)">
  <img id="bluePen" src="pen_blue.gif" alt="Blue Pen"
   onclick="changeColor('rgb(0,86,166)', this)">
</div>
```

The important part of this markup is the <img> element's onclick attribute. Whenever the web page visitor clicks a picture, the <img> element calls the changeColor() function. The changeColor() function accepts two pieces of information—the new color, which is set to match the icon, and a reference to the <img> element that was clicked. Here's the code:

```
// Keep track of the previous clicked <img> element for color.
var previousColorElement;

function changeColor(color, imgElement) {
  // Change the current drawing color.
  context.strokeStyle = color;

  // Give the newly clicked <img> element a new style.
  imgElement.className = "Selected";

  // Return the previously clicked <img> element to its normal state.
  if (previousColorElement != null) previousColorElement.className = "";
  previousColorElement = imgElement;
}
```

The changeColor() code takes care of two basic tasks. First, it sets the drawing context's strokeStyle property to match the new color; this takes a single line of code. Second, it changes the style of the clicked <img> element, giving it a solid border, so it's clear which color is currently active. This part requires a bit more work, because

the code needs to keep track of the previously selected color so it can remove the solid border around that <img> element.

The changeThickness() function performs an almost identical task, only it alters the lineWidth property to have the appropriate thickness:

```
// Keep track of the previously clicked <img> element for thickness.
var previousThicknessElement;


function changeThickness(thickness, imgElement) {
  // Change the current drawing thickness.
  context.lineWidth = thickness;

  // Give the newly clicked <img> element a new style.
  imgElement.className = "Selected";

  // Return the previously clicked <img> element to its normal state.
  if (previousThicknessElement != null) {
    previousThicknessElement.className = "";
  }
  previousThicknessElement = imgElement;
}
```

This code gets all the drawing setup out of the way, but this example still isn't ready to run. The next (and final) step is to add the code that performs the actual drawing.

Drawing on the Canvas

Drawing begins when the user clicks down on the canvas with the mouse button. The paint program uses a global variable named isDrawing to keep track of when drawing is taking place, so the rest of the code knows whether it should be writing on the drawing context.

As you saw earlier, the onMouseDown event is linked to the startDrawing() function. It sets the isDrawing variable, creates a new path, and then moves to the starting position to get ready to draw something:

```
var isDrawing = false;

function startDrawing(e) {
  // Start drawing.
  isDrawing = true;

  // Create a new path (with the current stroke color and stroke thickness).
  context.beginPath();

  // Put the pen down where the mouse is positioned.
  context.moveTo(e.pageX - canvas.offsetLeft, e.pageY - canvas.offsetTop);
}
```

In order for the paint program to work properly, it needs to start drawing at the current position—that's where the mouse is hovering when the user clicks down. However, getting the right coordinates for this point is a bit tricky.

The onMouseDown event provides coordinates (through the pageX and the pageY properties shown in this example), but these coordinates are relative to the whole page. To calculate the corresponding coordinate on the canvas, you need to subtract the distance between the top-left corner of the browser window and the top-left corner of the canvas.

The actual drawing happens while the user is moving the mouse. Every time the user moves the mouse, even just a single pixel, the onMouseMove event fires and the code in the draw() function runs. If isDrawing is set to true, then the code calculates the current canvas coordinate—where the mouse is *right now*—and then calls lineTo() to add a tiny line segment to the new position and stroke() to draw it:

```
function draw(e) {
  if (isDrawing == true) {
    // Find the new position of the mouse.
    var x = e.pageX - canvas.offsetLeft;
    var y = e.pageY - canvas.offsetTop;

    // Draw a line to the new position.
    context.lineTo(x, y);
    context.stroke();
  }
}
```

If the user keeps moving the mouse, the draw() function keeps getting called, and another short piece of line keeps getting added. This line is so short—probably just a pixel or two—that it doesn't even look like a straight line when the user starts scribbling.

Finally, when the user releases the mouse button or moves the cursor off to the side, away from the canvas, the onMouseUp or onMouseOut events fire. Both of these trigger the stopDrawing() function, which tells the application to stop drawing:

```
function stopDrawing() {
  isDrawing = false;
}
```

So far, this code covers almost all there is to the simple paint program. The only missing details are the two buttons under the canvas, which offer to clear or save the current work. Click clear, and the clearCanvas() function blanks out the entire surface, using the drawing context's clearRect() method:

```
function clearCanvas() {
  context.clearRect(0, 0, canvas.width, canvas.height);
}
```

The save option is slightly more interesting, and you'll consider it next.

Saving the Picture in the Canvas

When it comes to saving the picture in a canvas, there are countless options. First, you need to decide how you're going to get the data. The canvas gives you three basic options:

- **Use a data URL.** Converts the canvas to an image file and then converts that image data to a sequence of characters, formatted as a URL. This gives you a nice, portable way to pass the image data around (for example, you can hand it to an `<img>` element or send it off to the web server). The paint program uses this approach.
- **Use the getImageData() method.** Grabs the raw pixel data, which you can then manipulate as you please. You'll learn to use `getImageData()` on page 313.
- **Store a list of "steps."** Lets you store, for example, an array that lists every line you drew on the canvas. Then you can save that data and use it to recreate the image later. This approach takes less storage space, and it gives you more flexibility to work with or edit the image later on. Unfortunately, it works only if you keep track of all the steps you're taking, using a technique like the one you'll see in the circle-drawing example (page 294).

If that seems a bit intimidating, well, you're not done quite yet. Once you decide what you want to save, you still need to decide *where* to save it. Here are some options for that:

- **In an image file.** For example, you can let the web surfer save a PNG or JPEG on the computer. That's the approach you'll see next.
- **In the local storage system.** You'll learn how that works in Chapter 10.
- **On the web server.** Once you transfer the data, the web server could store it in a file or a database and make it available the next time the web page user visits.

To make the save feature work in the paint program, the code uses a feature called *data URLs*. To get a URL for the current data, you simply use the canvas's `toDataURL()` method:

```
var url = canvas.toDataURL();
```

When you call `toDataURL()` without supplying any arguments, you get a PNG-formatted picture. Alternatively, you can supply an image type to request that format instead:

```
var url = canvas.toDataURL("image/jpeg");
```

But if the browser is unable to honor your format request, it will still send you a PNG file, converted to a long string of letters.

At this point, you're probably wondering what a data URL looks like. Technically, it's a long string of base-64 encoded characters that starts with the text *data:image/png;base64*. It looks like gobbledygook, but that's OK, because it's supposed to be readable by computer programs (like browsers). Here's a data URL for the current canvas picture:


```

0IArs4c6QAAAARnQU1BAACxjwv8YQUAACqRSURBVHhe7Z1bkB1Hecdn5uxFFzA2FWOnsEEGiiew
nZgKsrWLrZXMRU9JgZQKhoSHVK...gAAEIQAACEIBAiAT+HxAYpeqDfKieAAAAAElFTkSuQmCC
```


This example leaves out a huge amount of the content in the middle (where the ellipsis is) to save space. If it was all left in, this data URL would fill five pages in this book.

NOTE Base-64 encoding is a system that converts image data to a long string of characters, numbers, and a small set of special characters. It avoids punctuation and all the bizarre extended characters, so the resulting text is safe to stick in a web page (for example, to set the value of a hidden input field or the `src` attribute in an `<img>` element).

So, it's easy to convert a canvas to image data in the form of a data URL. But once you have that data URL, what can you do with it? One option is to send it to the web server for long-term storage. You can see an example of a web page that does that, using a sprinkling of PHP script on the server, at *http://tinyurl.com/5uud9ob*.

If you want to keep your data on the client, your options are a bit more limited. Some browsers will let you navigate directly to a data URL. That means you can use code like the following to navigate to the image:

```
window.location = canvas.toDataURL();
```

A more reliable technique is to hand the data URL over to an `<img>` element. Here's what the paint program does (Figure 8-12):

```
function saveCanvas() {
  // Find the <img> element.
  var imageCopy = document.getElementById("savedImageCopy");

  // Show the canvas data in the image.
  imageCopy.src = canvas.toDataURL();

  // Unhide the <div> that holds the <img>, so the picture is now visible.
  var imageContainer = document.getElementById("savedCopyContainer");
  imageContainer.style.display = "block";
}
```

This code doesn't exactly "save" the image data, because the image hasn't yet been stored permanently, in a file. However, it takes just one more step to save the data once it's in an image. The web page visitor simply needs to right-click the image and choose the Save command. This isn't quite as convenient as a file download or the Save dialog box, but it's the only client-side option that works reliably in all browsers.

NOTE Firefox has the built-in ability to save canvas content. Just right-click any canvas (not the image copy) and choose Save Image As. Other browsers, like Chrome and Internet Explorer, don't offer this option.

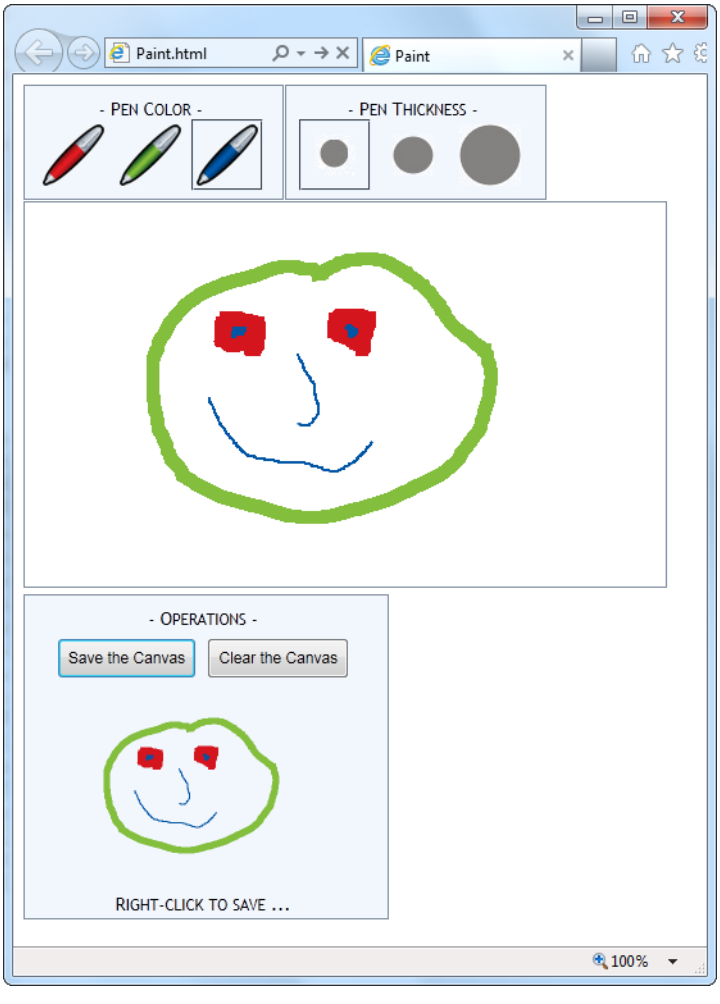

FIGURE 8-12

Here, the data URL is used to transmit information to an <img> element. The <img> element has been given a smaller size, to distinguish it from the main canvas. If you want to save this image as a respectable .png file, simply right-click the image and choose "Save picture as," just as you would with any other picture in a web page.

NOTE The data URL feature is one of several canvas features that may fail to work if you're running a test page from your computer hard drive. To avoid problems, upload your work to a live web server for testing.

GEM IN THE ROUGH

Canvas-Based Paint Programs

A paint program is one of the first examples that comes to mind when people start exercising their canvas-programming muscles. It's thus no surprise that you can Google up many more paint program examples on the Web, including some ridiculously advanced versions. Here are two favorites:

- **iPaint** (*http://tinyurl.com/js-ipaint*). This straightforward program looks like Microsoft Paint in a web browser. However, it adds at least one feature traditional Paint doesn't—the ability to select and manipulate the objects in your picture after you've drawn them.
- **Sketchpad** (*http://mugtug.com/sketchpad*). This amazingly tricked-out painting program has support for advanced illustration features like shape manipulation, marquee selection, textures, and even Spirograph drawing.

Browser Compatibility for the Canvas

You've come a long way with the canvas already. Now it's time to step back and answer the question that hangs over every new HTML5 feature: When is it safe to use it?

Fortunately, the canvas is one of the better-supported parts of HTML5. The latest version of every mainstream browser supports it. Of course, the more up-to-date the browser, the better—later builds of these browsers improve the drawing speed of the canvas and remove occasional glitches.

You're not likely to run into an old browser that doesn't support the canvas, except for versions of Internet Explorer before IE 9. And that's the clear issue that today's canvas users will be thinking about: How can you put the canvas in your pages without locking out IE 8 and IE 7?

As with many HTML5 features, you have two compatibility choices. Your first option is to detect when canvas support is missing and try to fall back on a different approach. Your second option is to fill the gap with another tool that can simulate the HTML5 canvas, so your pages will work—as written—on old browsers. In the case of the canvas, the second approach is the surprise winner, as you'll learn in the next section.

Polyfilling the Canvas with ExplorerCanvas

There are several solid workarounds that grant canvas-like abilities to aging copies of IE. The first is the ExplorerCanvas library (also called excanvas), by Google engineer and JavaScript genius Erik Arvidsson. It simulates the HTML5 canvas in Internet Explorer 7 or 8, using nothing but JavaScript and a now out-of-date technology called VML (Vector Markup Language).

NOTE VML is a specification for creating line art and other illustrations using markup in an HTML document. It's now been replaced by a similar, but better supported, standard called SVG (Scalable Vector Graphics), which browsers are just beginning to support. Today, VML still lingers in a few Microsoft products, like Microsoft Office and Internet Explorer. This makes it a reasonable substitute for the canvas, albeit with some limitations.

You can download ExplorerCanvas from *http://code.google.com/p/explorercanvas*. To use it, copy the *excanvas.js* file to the same folder as your web page, and add a script reference like this to your web page:

```
<head>
  <title>...</title>
  <!--[if lt IE 9]>
    <script src="excanvas.js"></script>
  <![endif]-->
  ...
<head>
```

Notice that this reference is conditional. Versions of Internet Explorer that are *earlier* than IE 9 will use it, but IE 9 and non-IE browsers will ignore the script altogether.

From this point, you can use the <canvas> without any serious headaches. For example, this change is all you need to get the basic paint program (shown on page 263) working in old versions of IE.

NOTE If you plan to draw text on your canvas (a feature discussed on page 279), you'll also need the help of a second JavaScript library, called Canvas-text, which works in conjunction with ExplorerCanvas. You can download it from *http://code.google.com/p/canvas-text*.

Polyfilling the Canvas with FlashCanvas

Of course, the ExplorerCanvas isn't perfect. If you use advanced features, you'll probably run into something that doesn't look right. The main features that aren't supported in ExplorerCanvas include radial gradients, shadows, clipping regions, raw pixel processing, and data URLs. And although someone may update ExplorerCanvas in the future to add these features, it doesn't seem likely—the current build of ExplorerCanvas is a few years old and the code hasn't been touched in a while.

If you have really ambitious plans—for example, you're planning to create complex animations or a side-scrolling game—you might find that ExplorerCanvas isn't fast enough to keep up. In this situation, you can consider switching to a different polyfill that uses a high-performance browser plug-in like Silverlight or Flash. You can review all your options on GitHub's polyfill page at *http://tinyurl.com/polyfills*. Or, go straight to one of the best: the free FlashCanvas library at *http://code.google.com/p/flashcanvas*. Like ExplorerCanvas, you can plug it into your page using a single line of JavaScript. But unlike ExplorerCanvas, it uses the Flash plug-in, without a trace of VML.

FlashCanvas also has a grown-up professional version called FlashCanvas Pro. It adds support for a few additional features, like global composite operations (page 262) and shadows (page 281).

FlashCanvas Pro is free for noncommercial use (get it at *http://flashcanvas.net/download*). Or, if you're a business or an individual planning to make some money, you can buy FlashCanvas Pro for a small fee (currently $31) at *http://flashcanvas.net/purchase*. You can compare the canvas support of ExplorerCanvas, FlashCanvas, and FlashCanvas Pro at *http://flashcanvas.net/docs/canvas-api*.

Like ExplorerCanvas, the FlashCanvas project isn't seeing much action these days. However, in its current state it remains a reliable choice for canvas support in old browsers. If you're planning to create a truly ambitious canvas-powered site—for example, a real-time game—you'll need to test out FlashCanvas to see if it supports everything you need to do.

NOTE When you combine a canvas-powered page with a library like FlashCanvas, you get truly impressive support on virtually every browser known to humankind. Not only do you get support for slightly older versions of IE through Flash, but you also get support for Flash-free mobile devices like the iPad and iPhone through HTML5.

The Canvas Fallback and Feature Detection

The most popular way to extend the reach of pages that use the canvas is with ExplorerCanvas and FlashCanvas. However, they aren't the only options.

The `<canvas>` element supports fallback content, just like the `<audio>` and `<video>` elements you explored in the last chapter. For example, you could apply this sort of markup to use the canvas (if it's supported) or to display an image (if it isn't):

```
<canvas id="logoCreator" width="500" height="300">
  <p>The canvas isn't supported on your computer, so you can't use our
  dynamic logo creator.</p>
  <img src="logo.png" alt="Standard Company Logo">
</canvas>
```

This technique is rarely much help. Most of the time, you'll use the canvas to draw dynamic graphics or to create some sort of interactive graphical application, and a fixed graphic just can't compensate. One alternative is to place a Flash application inside the `<canvas>` element. This approach is especially good if you already have a Flash version of your work but you're moving to the canvas for future development. It lets you offer a working Flash-based solution to old versions of IE, while letting everyone else use the plug-in-free canvas.

If you're using Modernizr (page 31), you can test for canvas support in your JavaScript code. Just test the `Modernizr.canvas` property, and check the `Modernizr.canvastext` property to look for the canvas's text-drawing feature (which was a later addition to the canvas drawing model). If you don't detect canvas support in the current browser, then you can use any workaround you'd like.

FREQUENTLY ASKED QUESTION

Canvas Accessibility

Is it possible to make the canvas more accessible?

One of the key themes of HTML5, the semantic elements, and the past few chapters is *accessibility*—designing a page that provides information to assistive software, so it can help people with disabilities use your website. After all this emphasis, it may come as a shock that one of HTML5's premier new features has no semantics or accessibility model *at all*.

The creators of HTML5 are working to patch the hole. However, no one is quite certain of the best solution. One proposal is to create a separate document model for assistive devices, which would then mirror the canvas content. The problem here is that it's up to the author to keep this "shadow" model in sync with the visuals, and lazy or overworked developers are likely to pass on this responsibility if it's at all complicated.

A second proposal is to extend image maps (an existing HTML feature that divides pictures into clickable regions) so they act as a layer over the top of the canvas. Because an image map is essentially a group of links, it could hold important information for assistive software to read and report to the user.

Currently, there's no point in thinking too much about either of these ideas, because they're both still being debated. In the meantime, it makes sense to use the canvas for a variety of graphical tasks, like arcade games (most of which can't practically be made accessible) and data visualization (as long as you have the data available in text form elsewhere on the page). However, the canvas isn't a good choice for an all-purpose page design element. So if you're planning to use the canvas to create a fancy heading or a menu for your website, hold off for now.

CHAPTER

9

Advanced Canvas: Interactivity and Animation

The canvas is a huge, sprawling feature. In the previous chapter, you learned how to draw line art and even create a respectable drawing program in a few dozen lines of JavaScript. But the canvas has more up its sleeve than that. Not only can it show dynamic pictures and host paint programs, but it can also play animations, process images with pixel-perfect control, and run interactive games. In this chapter, you'll learn the practical beginnings for all these tasks.

First, you'll start by looking at drawing context methods that let you paint different types of content on a canvas, including images and text. Next, you'll learn how to add some graphical pizzazz with shadows, patterned fills, and gradients. Finally, you'll learn practical techniques to make your canvas interactive and to host live animations. Best of all, you can build all of these examples with nothing more than ordinary JavaScript and raw ambition.

NOTE For the first half of this chapter, you'll focus on small snippets of drawing code. You can incorporate this code into your own pages, but you'll need to first add a `<canvas>` element to your page and create a drawing context, as you learned on page 246. In the second half of this chapter, you'll look at much more ambitious examples. Although you'll see most (or all) of the canvas-drawing code that these examples use, you won't get every page detail. To try out the examples for yourself, visit the try-out site at *http://prosetech.com/html5*.

Other Things You Can Draw on the Canvas

Using the canvas, you can painstakingly recreate any drawing you want, from a bunch of lines and triangles to a carefully shaded portrait. But as the complexity of

your drawing increases, so does the code. It's extremely unlikely that you'd write by hand all the code you need to create a finely detailed picture.

Fortunately, you have other options. The drawing context isn't limited to lines and curves—it also has methods that let you slap down pre-existing images, text, patterns, and even video frames. In the following sections, you'll learn how to use these methods to get more content on your canvas.

Drawing Images

You've probably seen web pages that build maps out of satellite images, downloaded and stitched together. That's an example of how you can take images you already have and combine them to get the final image that you want.

The canvas supports ordinary image data through the drawing context's logically named `drawImage()` method. To put an image in your canvas, you call `drawImage()` and pass in an image object and your coordinates, like this:

```
context.drawImage(img, 10, 10);
```

But before you can call `drawImage()`, you need the image object. HTML5 gives you three ways to get it. First, you can build it yourself out of raw pixels, one pixel at a time, using `createImageData()`. This approach is tedious and slow (although you'll learn more about per-pixel manipulation on page 313).

Your second option is to use an `<img>` element that's already on your page. For example, if you have this markup:

```
<img id="arrow_left" src="arrow_left.png">
```

You can copy that picture onto the canvas with this code:

```
var img = document.getElementById("arrow_left");
context.drawImage(img, 10, 10);
```

The third way that you can get an image for use with `drawImage()` is by creating an image object and loading an image picture from a separate file. The disadvantage to this approach is that you can't use your image with `drawImage()` until that picture has been completely downloaded. To prevent problems, you need to wait until the image's `onLoad` event occurs before you do anything with the image.

To understand this pattern, it helps to look at an example. Imagine you have an image named *maze.png* that you want to display on a canvas. Conceptually, you want to take this series of steps:

```
// Create the image object.
var img = new Image();

// Load the image file.
img.src = "maze.png";
```

```
// Draw the image. (This could fail, because the picture
// might not be downloaded yet.)
context.drawImage(img, 0, 0);
```

The problem here is that setting the `src` attribute starts an image download, but your code carries on without waiting for it to finish. The proper way to arrange this code is like this:

```
// Create the image object.
var img = new Image();

// Attach a function to the onload event.
// This tells the browser what to do after the image is loaded.
img.onload = function() {
  context.drawImage(img, 0, 0);
};

// Load the image file.
img.src = "maze.png";
```

This may seem counterintuitive, since the order in which the code is listed doesn't match the order in which it will be *executed*. In this example, the `context.drawImage()` call happens last, shortly after the `img.src` property is set.

Images have a wide range of uses. You can use them to add embellishments to your line drawings, or as a shortcut to avoid drawing by hand. In a game, you can use images for different objects and characters, positioned appropriately on the canvas. And fancy paint programs use them instead of basic line segments so the user can draw "textured" lines. You'll see some practical examples that use image drawing in this chapter.

TROUBLESHOOTING MOMENT

My Pictures Are Squashed!

If you attempt to draw a picture and find that it's inexplicably stretched, squashed, or otherwise distorted, the most likely culprit is a style sheet rule.

The proper way to size the canvas is to use its height and width attributes in your HTML markup. You might think you could remove these in your markup, leaving a tag like this:

```
<canvas></canvas>
```

And replace them with a style sheet rule that targets your canvas, like this one:

```
canvas {
  height: 300px;
  width: 500px;
}
```

But this doesn't work. The problem is that the CSS height and width properties aren't the same as the canvas height and width properties. If you make this mistake, what actually happens is that the canvas gets its default size (300 x 150 pixels). Then, the CSS size properties stretch or squash the canvas to fit, causing it to resize its contents accordingly. As a result, when you put an image on the canvas, it's squashed too, which is decidedly unappealing.

To avoid this problem, always specify the canvas size using its height and width attributes. And if you need a way to change the size of the canvas based on something else, use a bit of JavaScript code to change the `<canvas>` element's height and width when needed.

Slicing, Dicing, and Resizing an Image

The `drawImage()` function accepts a few optional arguments that let you alter the way your image is painted on the canvas. First, if you want to resize the image, you can tack on the width and height you want, like this:

```
context.drawImage(img, 10, 10, 30, 30);
```

This function makes a 30 x 30 pixel box for the image, with the top-left corner at point (10,10). Assuming the image is naturally 60 x 60 pixels, this operation squashes it by half in both dimensions, leaving it just a quarter as big as it would ordinarily be.

If you want to crop a piece out of the picture, you can supply the four extra arguments to `drawImage()` at the beginning of the argument list. These four points define the position and size of the rectangle you want to cut out of the picture, as shown here:

```
context.drawImage(img, source_x, source_y, source_width, source_height, x, y,
width, height);
```

The last four arguments are the same as in the previous example—they define the position and size that the cropped picture should have on the canvas.

For example, imagine you have a 200 x 200 pixel image and you want to paint just the top half. To do that, you create a box that starts at point (0,0) and has a width of 200 and a height of 100. You can then draw it on the canvas at point (75,25), using this code:

```
context.drawImage(img, 0, 0, 200, 100, 75, 25, 200, 100);
```

Figure 9-1 shows exactly what's happening in this example.

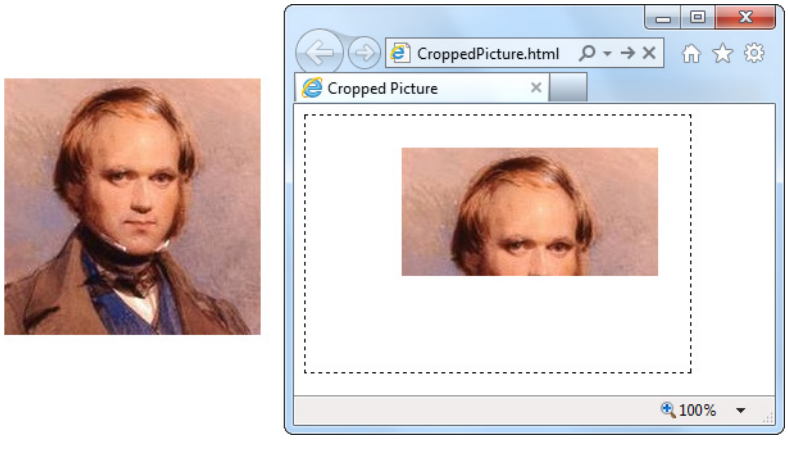

FIGURE 9-1

Left: The original source image.

Right: A cropped portion of the source image, on the canvas.

If you want to do more—for example, skew or rotate an image before you draw it, the `drawImage()` method can't keep up. However, you can use transforms to alter the way you draw anything and everything, as explained on page 256.

GEM IN THE ROUGH

Drawing a Video Frame

The first parameter of the `drawImage()` method is the image you want to draw. As you've seen, this can be an Image object you've just created, or an `<img>` element that's elsewhere on the page.

But that's not the whole story. HTML5 actually allows two more substitutions. Instead of an image, you can throw in a complete `<canvas>` element (not the one you're drawing on). Or, you can use a currently playing `<video>` element, with no extra work:

```
var video =
 document.getElementById("videoPlayer");

context.drawImage(video, 0, 0,
 video.clientWidth, video.clientWidth);
```

When this code runs, it grabs a single frame of video—the frame that's being played at the very instant the code runs. It then paints that picture onto the canvas.

This ability opens the door to a number of interesting effects. For example, you can use a timer to grab new video frames while playback is under way and keep painting them on a canvas. If you do this fast enough, the copied sequence of images on the canvas will look like another video player.

To get more exotic, you can change something about the copied video frame before you paint it. For example, you could scale it larger or smaller, or dip into the raw pixels and apply a Photoshop-style effect. For an example, read the article at *http://html5doctor.com/video-canvas-magic*. It shows how you can play a video in grayscale simply by taking regular snapshots of the real video and converting each pixel in each frame to a color-free shade of gray.

Drawing Text

Text is another thing that you wouldn't want to assemble yourself out of lines and curves. And the HTML5 canvas doesn't expect you to. Instead, it includes two drawing context methods that can do the job.

First, before you draw any text, you need to set the drawing context's `font` property. You use a string that uses the same syntax as the all-in-one CSS `font` property. At a bare minimum, you must supply the font size, in pixels, and the font name, like this:

```
context.font = "20px Arial";
```

You can supply a list of font names, if you're not sure that your font is supported:

```
context.font = "20px Verdana,sans-serif";
```

And optionally, you can add italics or bold at the beginning of the string, like this:

```
context.font = "bold 20px Arial";
```

You can also use a fancy web font, courtesy of CSS3. All you need to do is register the font name first, using a style sheet (as described on page 206).

Once the font is in place, you can use the `fillText()` method to draw your text. Here's an example that puts the top-left corner of the text at the point (10,10):

```
context.textBaseline = "top";
context.fillStyle = "black";
context.fillText("I'm stuck in a canvas. Someone let me out!", 10, 10);
```

You can put the text wherever you want, but you're limited to a single line. If you want to draw multiple lines of text, you need to call `fillText()` multiple times.

TIP If you want to divide a solid paragraph over multiple lines, you can create your own *word wrapping algorithm*. The basic idea is this: Split your sentence into words, and see how many words fit in each line using the drawing context's `measureText()` method. It's tedious to do, but the sample code at *http://tinyurl.com/6ec7hld* can get you started.

Instead of using `fillText()`, you can use the other text-drawing method, `strokeText()`. It draws an outline around your text, using the `strokeStyle` property for its color and the `lineWidth` property for its thickness. Here's an example:

```
context.font = "bold 40px Verdana,sans-serif";
context.lineWidth = "1";
context.strokeStyle = "red";
context.strokeText("I'm an OUTLINE", 20, 50);
```

When you use `strokeText()`, the middle of the text stays blank. Of course, you can use `fillText()` followed by `strokeText()` if you want colored, outlined text. Figure 9-2 shows both pieces of text in a canvas.

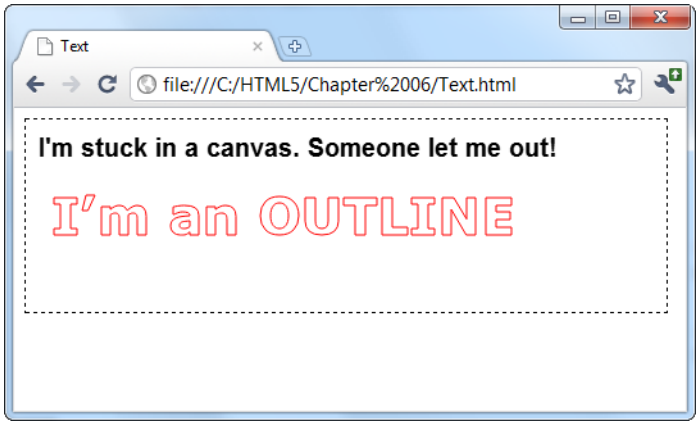

FIGURE 9-2
The canvas makes it easy to draw solid text and outlined text.

TIP Drawing text is much slower than drawing lines and images. The speed isn't important if you're creating a static, unchanging image (like a chart), but it may be an issue if you're creating an interactive, animated application. If you need to optimize performance, you may find that it's better to save your text in an image file and then draw it on the canvas with `drawImage()`.

Shadows and Fancy Fills

So far, when you've drawn lines and filled shapes on the canvas, you've used solid colors. And while there's certainly nothing wrong with that, ambitious painters will be happy to hear that the canvas has a few fancier drawing techniques. For example, the canvas can draw an artfully blurred shadow behind any shape. Or, it can fill a shape by tiling a small graphic across its surface. But the canvas's fanciest painting frill is *gradients*, which you can use to blend two or more colors into a kaleidoscope of patterns.

In the following sections, you'll learn to use all these features, simply by setting different properties in the canvas's drawing context.

Adding Shadows

One handy canvas feature is the ability to add a shadow behind anything you draw. Figure 9-3 shows some snazzy shadow examples.

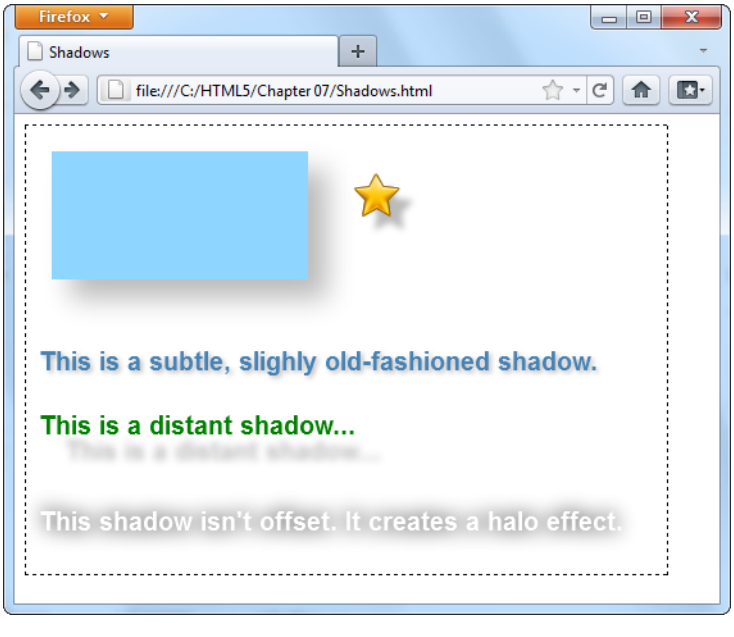

FIGURE 9-3

Shadows work equally well with shapes, images, and text. One nifty feature is the way that shadows work with images that have transparent backgrounds, like the star at the top right corner of this page. As you can see, the shadow follows the outline of the star shape, not the square box that delineates the entire image. (At the time of this writing, only Internet Explorer and Firefox support this feature.) Shadows also pair nicely with text, so you can create a range of different effects, depending on the shadow settings you pick.

Essentially, a shadow looks like a blurry version of what you would ordinarily draw (lines, shapes, images, or text). You control the appearance of shadows using several drawing context properties, as outlined in Table 9-1.

TABLE 9-1 *Properties for creating shadows*

PROPERTY	DESCRIPTION
shadowColor	Sets the shadow's color. You could go with black or a tinted color, but a nice midrange gray is generally best. Another good technique is to use a semitransparent color (page 185) so the content that's underneath still shows through. When you want to turn shadows off, set shadowColor back to transparent.
shadowBlur	Sets the shadow's "fuzziness." A shadowBlur of 0 creates a crisp shadow that looks just like a silhouette of the original shape. By comparison, a shadowBlur of 20 is a blurry haze, and you can go higher still. Most people agree that some fuzz (a blur of at least 3) looks best.
shadowOffsetX and shadowOffsetY	Positions the shadow relative to the content. For example, set both properties to 5, and the shadow will be bumped 5 pixels to the right and 5 pixels down from the original content. You can also use negative numbers to move the shadow the other way (left and up).

The following code creates the assorted shadows shown in Figure 9-3:

```
// Draw the shadowed rectangle.
context.rect(20, 20, 200, 100);
context.fillStyle = "#8ED6FF";
context.shadowColor = "#bbbbbb";
context.shadowBlur = 20;
context.shadowOffsetX = 15;
context.shadowOffsetY = 15;
context.fill();

// Draw the shadowed star.
context.shadowOffsetX = 10;
context.shadowOffsetY = 10;
context.shadowBlur = 4;
img = document.getElementById("star");
context.drawImage(img, 250, 30);

context.textBaseline = "top";
context.font = "bold 20px Arial";

// Draw three pieces of shadowed text.
context.shadowBlur = 3;
context.shadowOffsetX = 2;
context.shadowOffsetY = 2;
```

```
context.fillStyle = "steelblue";
context.fillText("This is a subtle, slightly old-fashioned shadow.", 10, 175);

context.shadowBlur = 5;
context.shadowOffsetX = 20;
context.shadowOffsetY = 20;
context.fillStyle = "green";
context.fillText("This is a distant shadow...", 10, 225);

context.shadowBlur = 15;
context.shadowOffsetX = 0;
context.shadowOffsetY = 0;
context.shadowColor = "black";
context.fillStyle = "white";
context.fillText("This shadow isn't offset. It creates a halo effect.", 10,
 300);
```

Filling Shapes with Patterns

So far, you've filled the shapes you've drawn with solid or semitransparent colors. But the canvas also has a fancy fill feature that lets you slather the inside with a pattern or a gradient. Using these fancy fills is a sure way to jazz up plain shapes. Using a fancy fill is a two-step affair. First, you create the fill. Then, you attach it to the `fillStyle` property (or, occasionally, the `strokeStyle` property).

To make a pattern fill, you start by choosing a small image that you can tile seamlessly over a large area (see Figure 9-4). You need to load the picture you want to tile into an image object using one of the techniques you learned about earlier, such as putting a hidden `<img>` on your page (page 276), or loading it from a file and handling the `onLoad` event of the `<img>` element (page 277). This example uses the first approach:

```
var img = document.getElementById("brickTile");
```

Once you have your image, you can create a pattern object using the drawing context's `createPattern()` method. At this point, you pick whether you want the pattern to repeat horizontally (`repeat-x`), vertically (`repeat-y`), or in both dimensions (`repeat`):

```
var pattern = context.createPattern(img, "repeat");
```

The final step is to use the pattern object to set the `fillStyle` or `strokeStyle` property:

```
context.fillStyle = pattern;
context.rect(0, 0, canvas.width, canvas.height);
context.fill();
```

This creates a rectangle that fills the canvas with the tiled image pattern, as shown in Figure 9-4.

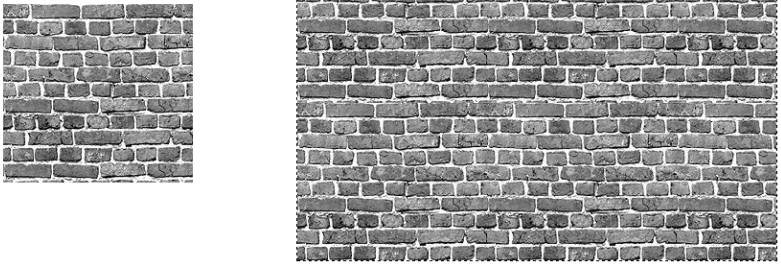

FIGURE 9-4

Left: An image that holds a single tile.

Right: The pattern created by tiling the image over an entire canvas.

Filling Shapes with Gradients

The second type of fancy fill is a gradient, which blends two or more colors. The canvas supports linear gradients and radial gradients, and Figure 9-5 compares the two.

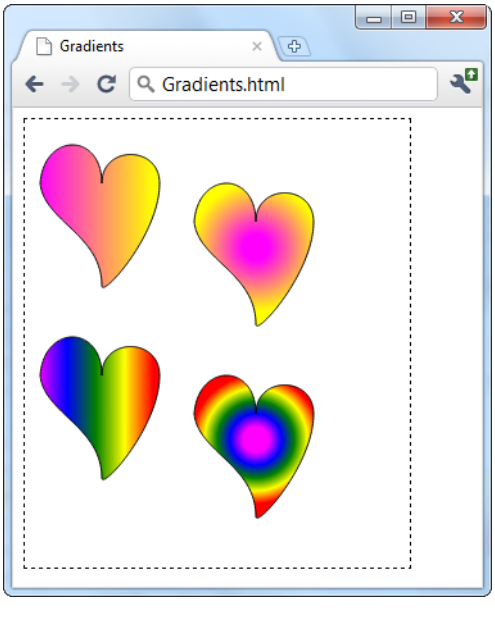

FIGURE 9-5

A linear gradient (top left) blends from one line of color to another. A radial gradient (top right) blends from one point of color to another. Both types support more than two colors, allowing you to create a banded effect with linear gradients (bottom left) or a ring effect with radial gradients (bottom right).

TIP If you're looking at these gradients in a black-and-white print copy of this book, give your head a shake and try out the examples at *http://prosetech.com/html5*, so you can see the wild colors for yourself. (The sample code also includes the drawing logic for the hearts, which stitches together four Bézier curves in a path.)

Unsurprisingly, the first step to using a gradient fill is creating the right type of gradient object. The drawing context has two methods that handle this task: `createLinearGradient()` and `createRadialGradient()`. Both work more or less the same way: They hold a list of colors that kick in at different points.

The easiest way to understand gradients is to start by looking at a simple example. Here's the code that's used to create the gradient for the top-left heart in Figure 9-5:

```
// Create a gradient from point (10,0) to (100,0).
var gradient = context.createLinearGradient(10, 0, 100, 0);

// Add two colors.
gradient.addColorStop(0, "magenta");
gradient.addColorStop(1, "yellow");

// Call another function to draw the shape.
drawHeart(60, 50);

// Paint the shape.
context.fillStyle = gradient;
context.fill();
context.stroke();
```

When you create a new linear gradient, you supply two points that represent the starting point and ending point of a *line*. This line is the path over which the color change takes place.

The gradient line is important, because it determines what the gradient looks like (see Figure 9-6). For example, consider a linear gradient that transitions from magenta to yellow. It could make this leap in a few pixels, or it could blend it out over the entire width of the canvas. Furthermore, the blend could be from left to right, top to bottom, or slanted somewhere in between. The line determines all these details.

TIP Think of a gradient as a colorful design just under the surface of your canvas. When you create a gradient, you're creating this colorful, but hidden, surface. When you fill a shape, you cut a hole that lets part of that gradient show through. The actual result—what you see in your canvas—depends on both the settings of your gradient and the size and position of your shape.

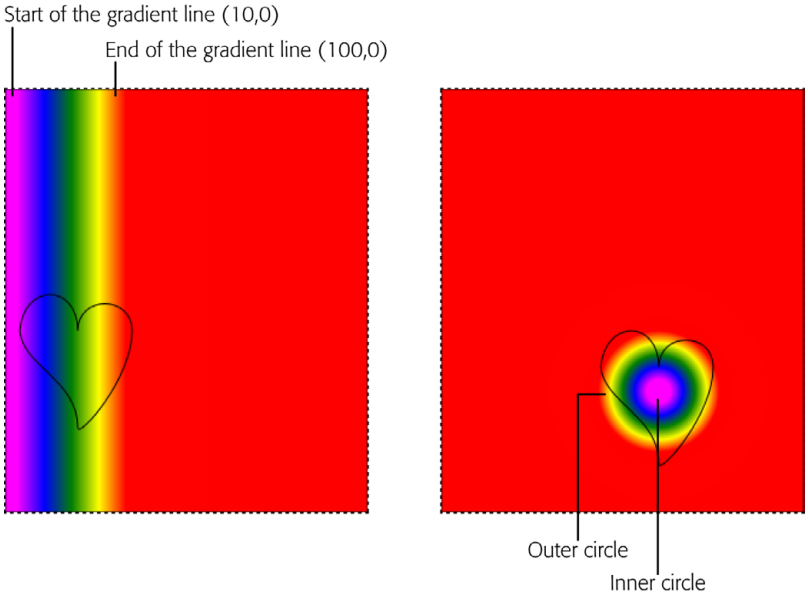

FIGURE 9-6

Left: Here's the gradient generated for the bottom-left heart from Figure 9-5. When you use that gradient to fill a shape, you see just a portion of it.

Right: The same is true for this radial gradient, which is used for the bottom-right heart in Figure 9-5.

In this example, the gradient line starts at point (10,0) and ends at point (100,0). These points tell you several important things:

- **The gradient is horizontal.** That means it blends colors from left to right. You know this because the two points have the same *y*-coordinate. If, on the other hand, you wanted to blend from top to bottom, you could use points like (0,10) and (0,100). If you wanted it to stretch diagonally from top left to bottom right, you could use (10,10) and (100,100).
- **The actual color blend spans just 90 pixels (starting when the *x*-coordinate is 10 and ending when the *x*-coordinate is 100).** In this example, the heart shape is just slightly smaller than the gradient dimensions, which means you see most of the gradient in the heart.
- **Beyond the limits of this gradient, the colors become solid.** So if you make the heart wider, you'll see more solid magenta (on the left) and solid yellow (on the right).

TIP Often, you'll create a gradient that's just barely bigger than the shape you're painting, as in this example. However, other approaches are possible. For example, if you want to paint several shapes using different parts of the same gradient, you might decide to create a gradient that covers the entire canvas.

To actually set the colors in a gradient, you call the gradient's `addColorStop()` method. Each time you do, you supply an offset from 0 to 1, which sets where the color appears in the blend. A value of 0 means the color appears at the very start of the gradient, while a value of 1 puts the color at the very end. Change these numbers (for example, to 0.2 and 0.8), and you can compress the gradient, exposing more solid color on either side.

When you create a two-color gradient, 0 and 1 make most sense for your offsets. But when you create a gradient with more colors, you can choose different offsets to stretch out some bands of colors while compressing others. The bottom-left heart in Figure 9-5 splits its offsets evenly, giving each color an equal-sized band:

```
var gradient = context.createLinearGradient(10, 0, 100, 0);
gradient.addColorStop("0", "magenta");
gradient.addColorStop(".25", "blue");
gradient.addColorStop(".50", "green");
gradient.addColorStop(".75", "yellow");
gradient.addColorStop("1.0", "red");

drawHeart(60, 200);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

NOTE If the room is starting to revolve around you, don't panic. After all, you don't need to understand everything that happens in a gradient. You can always just tweak the numbers until an appealing blend of colors appears in your shape.

You use the same process to create a radial gradient as you do to create a linear one. But instead of supplying two points, you must define two circles. That's because a radial gradient is a blend of color that radiates out from a small circle to a larger, containing circle. To define each of these circles, you supply the center point and radius.

In the radial gradient example shown in the top right of Figure 9-5, the colors blend from a center point inside the heart, at (180,100). The inner color is represented by a 10-pixel circle, and the outer color is a 50-pixel circle. Once again, if you go beyond these limits, you get solid colors, giving the radial gradient a solid magenta core and solid yellow surroundings. Here's the code that draws the two-color radial gradient:

```
var gradient = context.createRadialGradient(180, 100, 10, 180, 100, 50);
gradient.addColorStop(0, "magenta");
gradient.addColorStop(1, "yellow");

drawHeart(180, 80);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

NOTE Most often, you'll choose the same center point for both the inner and outer circles. However, you could offset one from the other, which can stretch, squash, and distort the blend of colors in interesting ways.

Using this example, you can create the final, multicolored radial gradient that's in the bottom-right corner of Figure 9-5. You simply need to move the center point of the circles to the location of the heart and add a different set of color stops—the same ones that you used for the multicolored linear gradient:

```
var gradient = context.createRadialGradient(180, 250, 10, 180, 250, 50);
gradient.addColorStop("0","magenta");
gradient.addColorStop(".25","blue");
gradient.addColorStop(".50","green");
gradient.addColorStop(".75","yellow");
gradient.addColorStop("1.0","red");

drawHeart(180, 230);
context.fillStyle = gradient;
context.fill();
context.stroke();
```

Now you have the smarts to create more psychedelic patterns than a 1960s revival party.

Putting It Together: Drawing a Graph

Now that you've slogged your way through the better part of the canvas's drawing features, it's time to pause and enjoy the fruits of your labor. In the following example, you'll take some humdrum text and numbers and use the canvas to create simple, standout charts.

Figure 9-7 shows the starting point: a two-page personality test that's light on graphics. The user answers the questions on the first page and then clicks Get Score to move to the second page. The second page reports the personality scores according to the infamous five-factor personality model (see the box on page 290).

The JavaScript code that makes this example work is pretty straightforward. When the user clicks a number button, its background is changed to indicate the user's choice. When the user completes the quiz, a simple algorithm runs the answers through a set of scoring formulas to calculate the five personality factors. If you want to examine this code or to take the test, visit the try-out site at *http://prosetech.com/html5*.

So far, there's no HTML5 magic. But consider how you could improve this two-page personality test by *graphing* the personality scores so that each factor is shown visually. Figure 9-8 shows the revamped version of the personality test results page, which uses this technique.

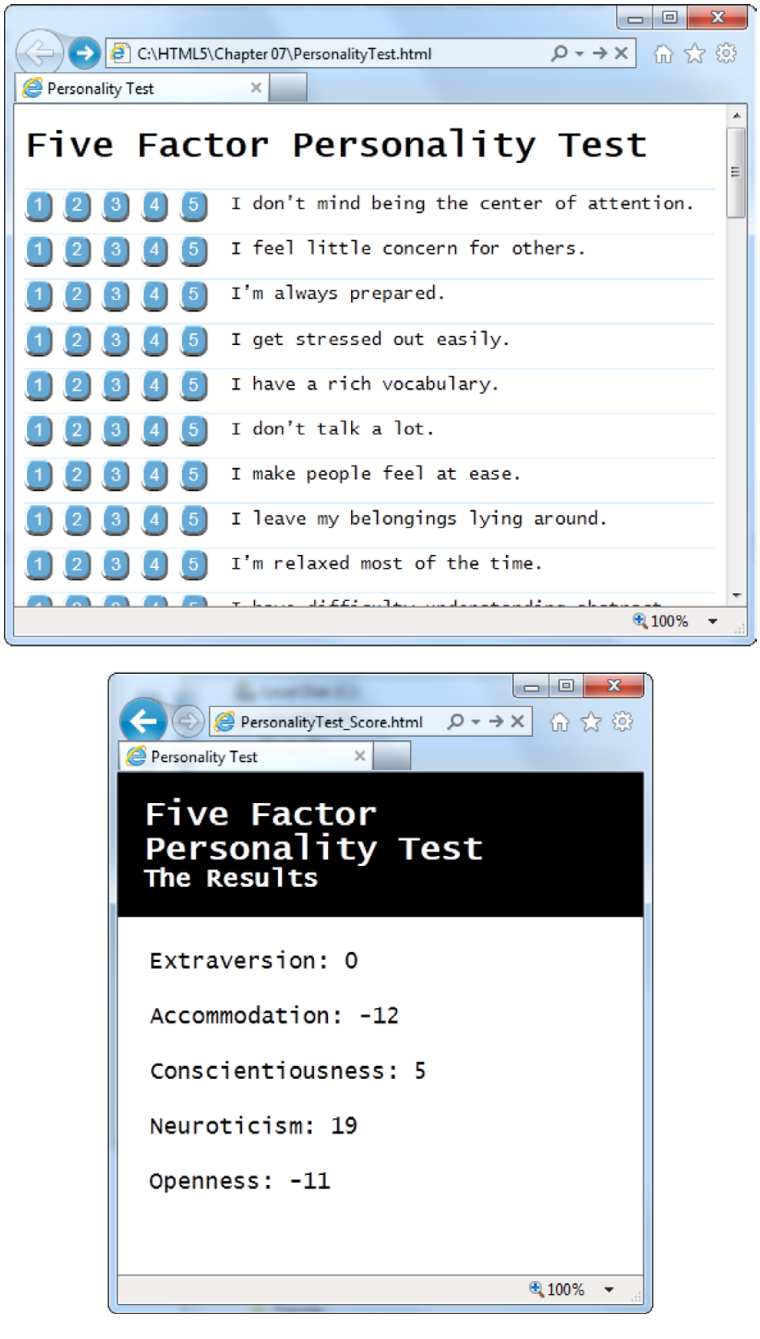

FIGURE 9-7

Click your way through the questions (top), and then review the scores (bottom). Unfortunately, without a scale or any kind of visual aid, it's difficult for ordinary people to tell what these numbers are supposed to mean.

UP TO SPEED

How to Convert One Personality into Five Numbers

This five-factor personality test ranks you according to five personality "ingredients," which are usually called openness, conscientiousness, extraversion, agreeableness, and neuroticism. These factors were cooked up when researchers analyzed the thousands of personality-describing adjectives in the English language.

To pick just five factors, psychologists used a combination of hard-core statistics, personality surveys, and a computer. They identified which adjectives people tend to tick off together and used that to distill the smallest set of personality super-traits.

For example, people who describe themselves as *outgoing* usually also describe themselves as *social* and *gregarious*, so it makes sense to combine all these traits into a single personality factor (which psychologists call *extraversion*). By the time the researchers had finished chewing through their set of nearly 20,000 adjectives, they had managed to boil them down to five closely interrelated factors.

You can learn more about the five-factor personality model at *http://tinyurl.com/big-five-p*, or in the book *Your Brain: The Missing Manual* (by this author).

To show these charts, the page uses five separate canvases, one for each personality factor. Here's the markup:

```
<header>
  <h1>Five Factor Personality Test</h1>
  <p>The Results</p>
</header>

<div class="score">
  <h2 id="headingE">Extraversion: </h2>
  <canvas id="canvasE" height="75" width="550"></canvas>
</div>

<div class="score">
  <h2 id="headingA">Accommodation: </h2>
  <canvas id="canvasA" height="75" width="550"></canvas>
</div>

<div class="score">
  <h2 id="headingC">Conscientiousness: </h2>
  <canvas id="canvasC" height="75" width="550"></canvas>
</div>

<div class="score">
  <h2 id="headingN">Neuroticism: </h2>
  <canvas id="canvasN" height="75" width="550"></canvas>
</div>
```

```
<div class="score">
  <h2 id="heading0">Openness: </h2>
  <canvas id="canvas0" height="75" width="550"></canvas>
</div>
```

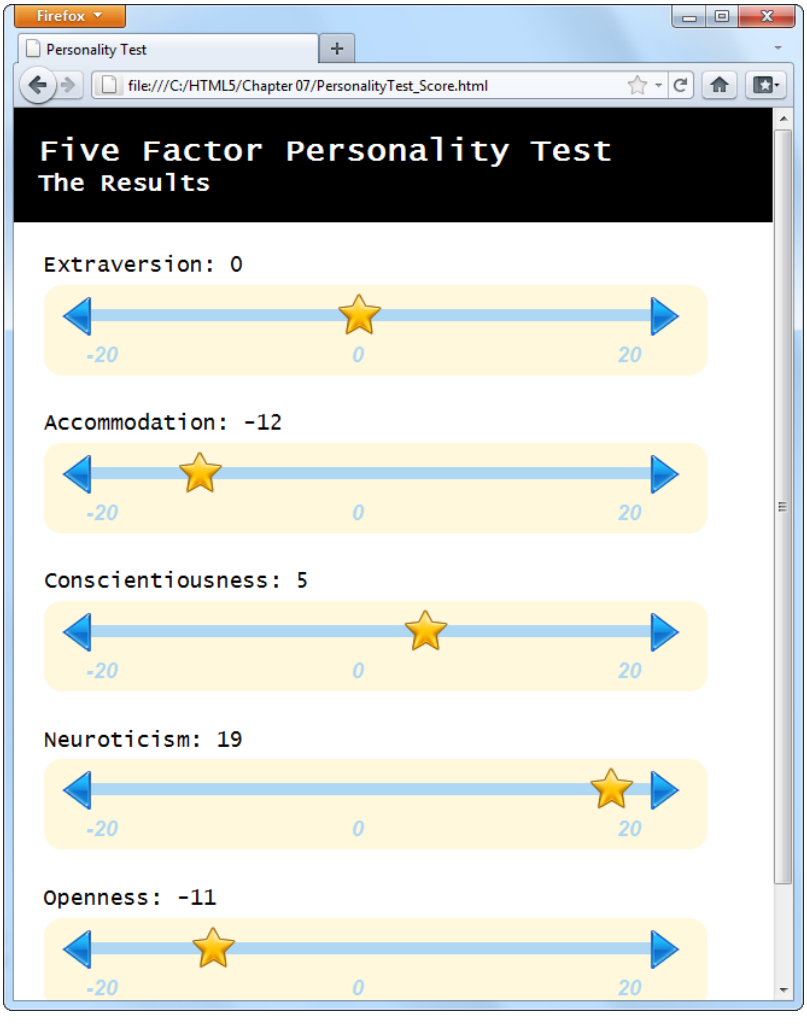

FIGURE 9-8

This page combines several different types of canvas drawing, including lines, images, and text. But the most interesting part of it is the way it dynamically draws these simple graphs, based on your answers in the personality quiz.

Each chart is drawn by the same custom JavaScript function, named `plotScore()`. The page calls this function five times, using different arguments each time. For example, to draw the extraversion chart at the top of the page, the code passes the topmost canvas element, the extraversion score (as a number from -20 to 20), and the text title ("Extraversion"):

```
window.onload = function() {
  ...
  // Get the <canvas> element for the extraversion score.
  var canvasE = document.getElementById("canvasE");

  // Add the score number to the corresponding heading.
  // (The score is stored in a variable named extraversion.)
  document.getElementById("headingE").innerHTML += extraversion;

  // Plot the score in the corresponding canvas.
  plotScore(canvasE, extraversion, "Extraversion");
  ...
}
```

The `plotScore()` function runs through a bit of drawing code that will seem familiar to you by now. It uses the various drawing context methods to draw the different parts of the score graph:

```
function plotScore(canvas, score, title) {
  var context = canvas.getContext("2d");

  // Draw the arrows on the side of the chart line.
  var img = document.getElementById("arrow_left");
  context.drawImage(img, 12, 10);
  img = document.getElementById("arrow_right");
  context.drawImage(img, 498, 10);

  // Draw the line between the arrows.
  context.moveTo(39, 25);
  context.lineTo(503, 25);
  context.lineWidth = 10;
  context.strokeStyle = "rgb(174,215,244)";
  context.stroke();

  // Write the numbers on the scale.
  context.fillStyle = context.strokeStyle;
  context.font = "italic bold 18px Arial";
  context.textBaseline = "top";

  context.fillText("-20", 35, 50);
  context.fillText("0", 255, 50);
  context.fillText("20", 475, 50);

  // Add the star to show where the score ranks on the chart.
  img = document.getElementById("star");
  context.drawImage(img, (score+20)/40*440+35-17, 0);
}
```

The most important bit is the final line, which plots the star at the right position using this slightly messy equation:

```
context.drawImage(img, (score+20)/40*440+35-17, 0);
```

Here's how it works. The first step is to convert the score into a percentage from 0 to 100 percent. Ordinarily, the score falls between -20 and 20, so the first operation the code needs to carry out is to change it to a value from 0 to 40:

```
score+20
```

You can then divide that number by 40 to get the percentage:

```
(score+20)/40
```

Once you have the percentage, you need to multiply it by the length of the line. That way, 0 percent ends up at the far left side, while 100 percent ends up at the opposite end, and everything else falls somewhere in between:

```
(score+20)/40*440
```

This code would work fine if the line stretched from the *x*-coordinate 0 to the *x*-coordinate 400. But in reality the line is offset a bit from the left edge, to give it a bit of padding. You need to offset the star by the same amount:

```
(score+20)/40*440+35
```

But this lines the left edge of the start up with the proper position, when really you want to line up its midpoint. To correct this issue, you need to subtract an amount that's roughly equal to half the start's width:

```
(score+20)/40*440+35-17
```

This gives you the final *x*-coordinate for the star, based on the score.

NOTE It's a small jump to move from fixed drawings to dynamic graphics like the ones in this example, which tailor themselves according to the latest data. But once you've made this step, you can apply your skills to build all sorts of interesting data-driven graphics, from traditional pie charts to infographics that use dials and meters. And if you're looking for a way to simplify the task, check out one of the canvas graphic libraries, which include pre-written JavaScript routines for drawing common types of graphs based on your data. Two good examples are RGraph (*www.rgraph.net*) and ZingChart (*www.zingchart.com*).

Making Your Shapes Interactive

The canvas is a *non-retained* painting surface. That means that the canvas doesn't keep track of what drawing operations you've performed. Instead, it just keeps the final result—the grid of colored pixels that makes up your picture.

For example, if you draw a red square in the middle of your canvas, the moment you call `stroke()` or `fill()`, that square becomes nothing more than a block of red pixels. It may look like a square, but the canvas has no memory of its squareness.

This model makes drawing fast. However, it also makes life difficult if you want to add interactivity to your drawing. For example, imagine you want to create a smarter version of the painting program you saw on page 263. You want people to be able to not only draw a line, but also draw a rectangle. (That part's easy.) And you want people to be able to not only draw a rectangle, but also select it, drag it to a new place, resize it, change its color, and so on. Before you can give them all that power, you need to deal with several complications. First, how do you know if someone has clicked on the rectangle? Second, how do you know the details about the rectangle—its coordinates, size, stroke color, and fill color? And third, how do you know the details about every *other* shape on the canvas—which you'll need to know if you need to change the rectangle and repaint the canvas?

To solve these problems and make your canvas interactive, you need to keep track of every object you paint. Then, when users click somewhere, you need to check whether they're clicking one of the shapes (a process known as *hit testing*). If you can tackle these two tasks, the rest—changing one of your shapes and repainting the canvas—is easy.

Keeping Track of What You've Drawn

In order to be able to change and repaint your picture, you need to know everything about its contents. For example, consider the circle-drawing program shown in Figure 9-9. To keep things simple, it consists entirely of separately sized, differently colored circles.

To keep track of an individual circle, you need to know its position, radius, and fill color. Rather than create dozens of variables to store all this information, it makes sense to put all four details in a small package. That package is a *custom object*.

If you haven't created a custom object before, here's the standard technique. First, you create a function that's named after your type of object. For example, if you want to build a custom object for a circle creation, you might name the function `Circle()`, like this:

```
function Circle() {
}
```

Next, you need your object to store some data. You do that by using a keyword named `this` to create properties. For example, if you want to give your circle object a property named `radius`, so you can keep track of its size, you would assign a starting value to `this.radius` inside the `Circle()` function.

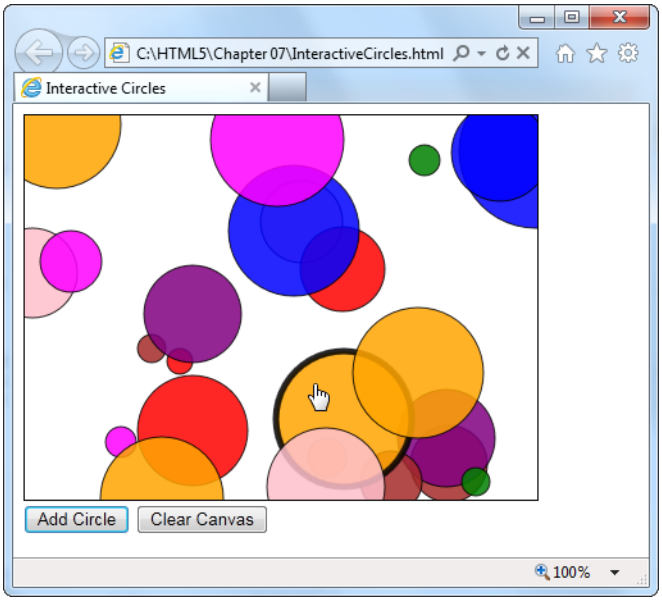

FIGURE 9-9

This circle-drawing program is interactive. You can click to select a circle (at which point it becomes highlighted with a different border color) and drag it to a new position.

Here's a function that defines a circle with three details: the *x*- and *y*-coordinates, and the radius of the circle:

```
function Circle() {
  this.x = 0;
  thix.y = 0;
  this.radius = 15;
}
```

The numbers you set in the `Circle()` function are just default values. When you use the function to create a new circle object, you can change each property value.

Once you've added all the properties you need, you're ready to use your function to create live objects. In this case, that means using the `Circle()` function to create a new circle object. The trick is that you don't want to call the function—instead, you want to create a new copy of it with the `new` keyword. Here's an example:

```
// Create a new circle object, and store it in a variable named myCircle.
var myCircle = new Circle();
```

Once you have a live circle object, you can access all the circle details as properties:

```
// Change the radius.
myCircle.radius = 20;
```

If you want to get a bit fancier, you can pass arguments to the `Circle()` function. That way, you can create a circle object and set all the circle properties in one step. Here's the version of the `Circle()` function that's used for the page in Figure 9-9:

```
function Circle(x, y, radius, color) {
  this.x = x;
  this.y = y;
  this.radius = radius;
  this.color = color;
  this.isSelected = false;
}
```

The `isSelected` property takes a `true` or `false` value. When someone clicks the circle, `isSelected` is set to `true`, and then the drawing code knows to paint a different border around it.

Using this version of the `Circle()` function, you can use code like this to create a circle object:

```
var myCircle = new Circle(0, 0, 20, "red");
```

Of course, the whole point of the circle-drawing program is to let people draw as many circles as they want. That means just one circle object won't do. Instead, you need to create an array that can hold all the circles. Here's the global variable that makes this work in the current example:

```
var circles = [];
```

The rest of the code is easy. When someone clicks the Add Circle button to create a new circle, it triggers the `addRandomCircle()` function. The `addRandomCircle()` function creates a new circle with a random size, color, and position:

```
function addRandomCircle() {
  // Give the circle a random size and position.
  var radius = randomFromTo(10, 60);
  var x = randomFromTo(0, canvas.width);
  var y = randomFromTo(0, canvas.height);

  // Give the circle a random color.
  var colors = ["green", "blue", "red", "yellow", "magenta",
   "orange", "brown", "purple", "pink"];
  var color = colors[randomFromTo(0, 8)];

  // Create the new circle.
  var circle = new Circle(x, y, radius, color);

  // Store it in the array.
  circles.push(circle);
```

```
    // Redraw the canvas.
    drawCircles();
  }
```

This code makes use of a custom function named `randomFromTo()`, which generates random numbers in a set range.

(To play with the full code, visit *http://prosetech.com/html5*.)

The final step in this sequence is actually painting the canvas, based on the current collection of circles. After a new circle is created, the `addRandomCircle()` function calls another function, named `drawCircles()`, to do the job. The `drawCircles()` function moves through the array of circles, using this loop:

```
for(var i=0; i<circles.length; i++) {
  var circle = circles[i];
  ...
}
```

This code uses the trusty `for` loop (which is described in Appendix B, "JavaScript: The Brains of Your Page"). The block of code in the braces runs once for each circle. The first line of code grabs the current circle and stores it in a variable, so it's easy to work with.

Here's the complete `drawCircles()` function, which fills the canvas with the current collection of circles:

```
function drawCircles() {
  // Clear the canvas and prepare to draw.
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.globalAlpha = 0.85;
  context.strokeStyle = "black";

  // Go through all the circles.
  for(var i=0; i<circles.length; i++) {
    var circle = circles[i];

    // Draw the circle.
    context.beginPath();
    context.arc(circle.x, circle.y, circle.radius, 0, Math.PI*2);
    context.fillStyle = circle.color;

    context.fill();
    context.stroke();
  }
}
```

NOTE Each time the circle-drawing program refreshes the display, it starts by clearing everything away with the `clearRect()` method. Paranoid programmers might worry that this step would cause a flicker, as the canvas goes blank and the circles disappear, then reappear. However, the canvas is optimized to prevent this problem. It doesn't actually clear or paint *anything* until all your drawing logic has finished, at which point it copies the final product to the canvas in one smooth step.

Right now, the circles still aren't interactive. However, the page has the all-important plumbing that keeps track of every circle that's drawn. Although the canvas is still just a block of colored pixels, the code knows the precise combination of circles that creates the canvas—and that means it can manipulate those circles at any time.

In the next section, you'll see how you can use this system to let the user select a circle.

Hit Testing with Coordinates

If you're creating interactive shapes, you'll almost certainly need to use *hit testing*, a technique that checks whether a given point has "hit" another shape. In the circle-drawing program, you want to check whether the user's click has hit upon a circle or just empty space.

More sophisticated animation frameworks (like those provided by Flash and Silverlight) handle hit testing for you. And there are add-on JavaScript libraries for the canvas (like KineticJS) that aim to offer the same conveniences, but none is mature enough to recommend without reservations. So the best bet is for canvas fans to start by learning to write their own hit testing logic. (After you've done that, you can consider pumping up the canvas with one of the JavaScript libraries discussed in the box on page 307.)

To perform hit testing, you need to examine each shape and calculate whether the point is inside the bounds of that shape. If it is, the click has "hit" that shape. Conceptually, this process is straightforward, but the actual details—the calculations you need to figure out whether a shape has been clicked—can get messy.

The first thing you need is a loop that moves through all the shapes. This loop looks a little different from the one that the `drawCircles()` function uses:

```
for (var i=circles.length-1; i>=0; i--) {
  var circle = circles[i];
  ...
}
```

Notice that the code actually moves backward through the array, from finish to start. It starts at the end (where the index is equal to the total number of items in the array, minus 1), and counts back to the beginning (where the index is 0). The backward looping is deliberate, because in most applications (including this one), the objects are drawn in the same order they're listed in the array. That means later objects are superimposed over earlier ones, and when shapes overlap, it's always the shape on top that should get the click.

To determine if a click has landed in a shape, you need to use some math. In the case of a circle, you need to calculate the straight-line distance from the clicked point to the center of the circle. If the distance is less than or equal to the radius of the circle, then the point lies inside its bounds.

In the current example, the web page handles the `onClick` event of the canvas to check for circle clicks. When the user clicks the canvas, it triggers a function named `canvasClick()`. That function figures out the click coordinates and then sees if they intersect any circle:

```
function canvasClick(e) {
  // Get the canvas click coordinates.
  var clickX = e.pageX - canvas.offsetLeft;
  var clickY = e.pageY - canvas.offsetTop;

  // Look for the clicked circle.
  for (var i=circles.length; i>0; i--) {
    // Use Pythagorean theorem to find the distance from this point
    // and the center of the circle.
    var distanceFromCenter =
    Math.sqrt(Math.pow(circle.x - clickX, 2) + Math.pow(circle.y - clickY, 2))

    // Does this point lie in the circle?
    if (distanceFromCenter <= circle.radius) {
      // Clear the previous selection.
      if (previousSelectedCircle != null) {
        previousSelectedCircle.isSelected = false;
      }
      previousSelectedCircle = circle;

      // Select the new circle.
      circle.isSelected = true;

      // Update the display.
      drawCircles();

      // Stop searching.
      return;
    }
  }
}
```

NOTE You'll look at another way to do hit testing—by grabbing raw pixels and checking their color—when you create a maze game, on page 313.

To finish this example, the drawing code in the `drawCircles()` function needs a slight tweak. Now it needs to single out the selected circle for special treatment (in this case, a thick, bold border):

```
function drawCircles() {
  ...

  // Go through all the circles.
  for(var i=0; i<circles.length; i++) {
    var circle = circles[i];

    if (circle.isSelected) {
      context.lineWidth = 5;
    }
    else {
      context.lineWidth = 1;
    }
    ...
  }
}
```

There's no end of ways that you can build on this example and make it smarter. For example, you can add a toolbar of commands that modify the selected circle—for example, changing its color or deleting it from the canvas. Or, you can let the user drag the selected circle around the canvas. To do that, you simply need to listen for the `onMouseMove` event of the canvas, change the circle coordinates accordingly, and call the `drawCircles()` function to repaint the canvas. (This technique is essentially a variation of the mouse-handling logic in the paint program on page 263, except now you're using mouse movements to drag a shape, not draw a line.) The try-out site (*http://prosetech.com/html5*) includes a variation of this example named *InteractiveCircles_WithDrag.html* that demonstrates this technique.

The lesson is clear: If you keep track of what you draw, you have unlimited flexibility to change it and redraw it later on.

Animating the Canvas

Drawing one perfect picture can seem daunting enough. So it's no surprise that even seasoned web developers approach the idea of drawing several dozen each second with some trepidation. The whole challenge of animation is to draw and redraw canvas content fast enough that it looks like it's moving or changing right in front of your visitors.

Animation is an obvious and essential building block for certain types of applications, like real-time games and physics simulators. But simpler forms of animation make sense in a much broader range of canvas-powered pages. You can use animation to highlight user interactivity (for example, making a shape glow, pulse, or twinkle when

someone hovers over it). You can also use animation to draw attention to changes in content (for example, fading in a new scene or creating graphs and charts that "grow" into the right positions). Used in these ways, animation is a powerful way to give your web applications some polish, make them feel more responsive, and even help them stand out from the crowd of web competitors.

A Basic Animation

It's easy enough to animate a drawing on an HTML5 canvas. First, you set a timer that calls your drawing over and over again—typically 30 or 40 times each second. Each time, your code repaints the entire canvas from scratch. If you've done your job right, the constantly shifting frames will blend into a smooth, lifelike animation.

JavaScript gives you two ways to manage this repetitive redrawing:

- **Use the `setTimeout()` function.** This tells the browser to wait a few milliseconds and then run a piece of code—in this case, that's your canvas-drawing logic. Once your code finishes, you call `setTimeout()` to ask the browser to call it again, and so on, until you want your animation to end.
- **Use the `setInterval()` function.** Tells your browser to run a piece of code at regular intervals (say, every 20 milliseconds). It has much the same effect as `setTimeout()`, but you need to call `setInterval()` only once. To stop the browser from calling your code, you call `clearInterval()`.

If your drawing code is quick, both of these have the same effect. If your drawing code is less than snappy, then the `setInterval()` approach will do a better job of keeping your redraws precisely on time, but possibly at the expense of performance. (In the worst-case situation, when your drawing code takes a bit longer than the interval you've set, your browser will struggle to keep up, your drawing code will run continuously, and your page may briefly freeze up.) For this reason, the examples in this chapter use the `setTimeout()` approach.

When you call `setTimeout()`, you supply two pieces of information: the name of the function you want to run, and the amount of time to wait before running it. You give the amount of time as a number of milliseconds (thousandths of a second), so 20 milliseconds (a typical delay for animation) is 0.02 seconds. Here's an example:

```
var canvas;
var context;

window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  // Draw the canvas in 0.02 seconds.
  setTimeout(drawFrame, 20);
};
```

This call to `setTimeout()` is the heart of any animation task. For example, imagine you want to make a square fall from the top of the page to the bottom. To do this, you need to keep track of the square's position using two global variables:

```
// Set the square's initial position.
var squarePosition_y = 0;
var squarePosition_x = 10;
```

Now, you simply need to change the position each time the `drawFrame()` function runs, and then redraw the square in its new position:

```
function drawFrame() {
  // Clear the canvas.
  context.clearRect(0, 0, canvas.width, canvas.height);

  // Call beginPath() to make sure you don't redraw
  // part of what you were drawing before.
  context.beginPath();

  // Draw a 10x10 square, at the current position.
  context.rect(squarePosition_x, squarePosition_y, 10, 10);
  context.lineStyle = "black";
  context.lineWidth = 1;
  context.stroke();

  // Move the square down 1 pixel (where it will be drawn
  // in the next frame).
  squarePosition_y += 1;

  // Draw the next frame in 20 milliseconds.
  setTimeout(drawFrame, 20);
}
```

Run this example, and you'll see a square that plummets from the top of the canvas and carries on, disappearing past the bottom edge.

In a more sophisticated animation, the calculations get more complex. For example, you may want to make the square accelerate to simulate gravity, or bounce at the bottom of the page. But the basic technique—setting a timer, calling a drawing function, and redrawing the entire canvas—stays exactly the same.

Animating Multiple Objects

Now that you've learned the basics of animation and the basics of making interactive canvas drawings, it's time to take the next step and merge them together. Figure 9-10 shows a test page with an animation of falling, bouncing balls. The page uses the familiar `setTimeout()` method you met in the last section, but now the drawing code has to manage an essentially unlimited number of flying balls.

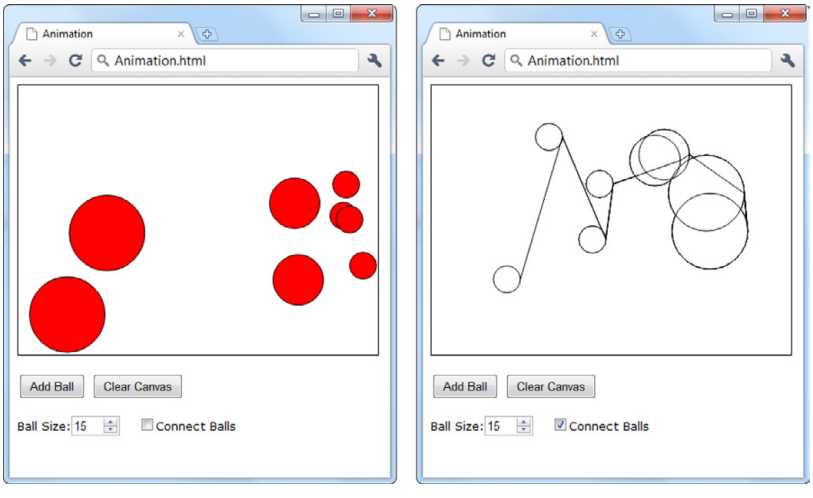

FIGURE 9-10

This test page lets you add as many balls as you want. You can choose the size of each ball (the default is a 15-pixel radius), and you can switch on connecting lines (shown on right). Once added, each ball begins its own independent journey, falling down and picking up speed until it collides with the bottom of the canvas and bounces off somewhere else.

TIP Pictures can't do justice to animations. To find out how animations like the one in Figure 9-10 look and feel, you can try out the examples for yourself at *http://prosetech.com/html5*.

UP TO SPEED

Animation Performance

Because of these rapid redraws, it's clear that animation will tax the abilities of the canvas far more than ordinary drawing does. However, the canvas holds up surprisingly well. Modern browsers use performance-enhancing features like *hardware acceleration*, which farms out some of the graphical work to the computer's video card, rather than using its CPU for everything. And even though JavaScript isn't the fastest language on the block, it's quite possible to create a complex, high-speed animation—even a real-time arcade game—without using anything except script code and the canvas.

However, canvas performance *can* become a problem for people using underpowered mobile devices, like iPhones or phones running Google's Android operating system. Tests have shown that an animation that can run at 60 fps (frames per second) on a desktop browser probably tops out at a jerky 10 fps on a smartphone. So if you want to create an application for mobile visitors, make sure you test before you get too far into your design, and be prepared to sacrifice some of the animation eye candy to keep things running smoothly.

To manage all the balls in Figure 9-10, you need to reuse the custom object trick you picked up on page 295. Only now you need to track an array of ball objects, and each ball needs not only a position (represented by the properties x and y), but also a speed (represented by dx or dy):

```
// These are the details that represent an individual ball.
function Ball(x, y, dx, dy, radius) {
  this.x = x;
  this.y = y;
  this.dx = dx;
  this.dy = dy;
  this.radius = radius;
  this.color = "red";
}

// This is an array that will hold all the balls on the canvas.
var balls = [];
```

NOTE In mathematics lingo, dx is the rate that x is changing, and dy is the rate that y is changing. So as the ball is falling, each frame x increases by the dx amount and y increases by the dy amount.

When the visitor clicks the Add Ball button, a simple bit of code creates a new ball object and stores it in the `balls` array:

```
function addBall() {
  // Get the requested size.
  var radius = parseFloat(document.getElementById("ballSize").value);

  // Create the new ball.
  var ball = new Ball(50,50,1,1,radius);

  // Store it in the balls array.
  balls.push(ball);
}
```

The Clear Canvas button has the complementary task of emptying the `balls` array:

```
function clearBalls() {
  // Remove all the balls.
  balls = [];
}
```

However, neither the `addBall()` nor the `clearBalls()` function actually draws anything. Neither one even calls a drawing function. Instead, the page sets itself up to call the `drawFrame()` function, which paints the canvas in 20 milliseconds intervals:

```
var canvas;
var context;

window.onload = function() {
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");
```

```
  // Redraw every 20 milliseconds.
  setTimeout(drawFrame, 20);
};
```

The `drawFrame()` function is the heart of this example. It not only paints the balls on the canvas, but it also calculates their current position and speed. The `drawFrame()` function uses a number of calculations to simulate more realistic movement—for example, making balls accelerate as they fall and slow down when they bounce off of obstacles. Here's the complete code:

```
function drawFrame()   // Clear the canvas.
  context.clearRect(0, 0, canvas.width, canvas.height);
  context.beginPath();

  // Go through all the balls.
  for(var i=0; i<balls.length; i++) {
    // Move each ball to its new position.
    var ball = balls[i];
    ball.x += ball.dx;
    ball.y += ball.dy;

    // Add in a "gravity" effect that makes the ball fall faster.
    if ((ball.y) < canvas.height) ball.dy += 0.22;

    // Add in a "friction" effect that slows down the side-to-side motion.
    ball.dx = ball.dx * 0.998;

    // If the ball has hit the side, bounce it.
    if ((ball.x + ball.radius > canvas.width) || (ball.x - ball.radius < 0)) {
      ball.dx = -ball.dx;
    }

    // If the ball has hit the bottom, bounce it, but slow it down slightly.
    if ((ball.y + ball.radius > canvas.height) || (ball.y - ball.radius < 0))
    {
      ball.dy = -ball.dy*0.96;
    }

    // Check if the user wants lines.
    if (!document.getElementById("connectedBalls").checked) {
      context.beginPath();
      context.fillStyle = ball.fillColor;
    }
    else {
      context.fillStyle = "white";
    }
```

```
        // Draw the ball.
        context.arc(ball.x, ball.y, ball.radius, 0, Math.PI*2);
        context.lineWidth = 1;
        context.fill();
        context.stroke();
      }

      // Draw the next frame in 20 milliseconds.
      setTimeout(drawFrame, 20);
    }
```

TIP If you're fuzzy about how the `if` statements work in this example, and what operators like `!` and `||` really mean, check out the summary of logical operators on page 463 in Appendix B, "JavaScript: The Brains of Your Page."

The sheer amount of code can seem a bit intimidating. But the overall approach hasn't changed. The code performs the same steps:

1. Clear the canvas.
2. Loop through the array of balls.
3. Adjust the position and velocity of each ball.
4. Paint each ball.
5. Set a timeout so the `drawFrame()` method will be called again, 20 milliseconds later.

The complex bit is step 3, where the ball is tweaked. This code can be as complicated as you like, depending on the effect you're trying to achieve. Gradual, natural movement is particularly difficult to model, and it usually needs more math.

Finally, now that you've done all the work tracking each ball, it's easy to add interactivity. In fact, you can use virtually the same code you used to detect clicks in the circle-drawing program on page 298. Only now, when a ball is clicked, you want something else to happen—for example, you might choose to give the clicked ball a sudden boost in speed, sending it ricocheting off to the side. (The downloadable version of this example, available at *http://prosetech.com/html5*, does exactly that.)

To see this example carried to its most impressive extreme, check out the bouncing Google balls at *http://tinyurl.com/6byvnk5*. When left alone, the balls are pulled, magnet-like, to spell the word "Google." But when your mouse moves in, they're repulsed, flying off to the far corners of the canvas and bouncing erratically. And if you're still hungry for more animation examples, check out the poke-able blob at *http://www.blobsallad.se* and the somewhat clichéd flying star field at *http://tinyurl.com/crn3ed*.

FREQUENTLY ASKED QUESTION

Canvas Animations for Busy (or Lazy) People

Do I really need to calculate everything on my own? For real?

The most significant drawback to canvas animation is the fact that you need to do everything yourself. For example, if you want a picture to fly from one side of the canvas to the other, you need to calculate its new position in each frame, and then draw the picture in its proper location. If you have several things being animated at the same time in different ways, your logic can quickly get messy. By comparison, life is much easier for programmers who are using a browser plug-in like Flash or Silverlight. Both technologies have built-in animation systems, which allow developers to give instructions like "Move this shape from here to there, taking 45 seconds." Or, even better, "Move this shape from the top of the window to the bottom, using an accelerating effect that ends with a gentle bounce."

To fill this gap, enterprising JavaScript developers have begun creating higher-level drawing and animation systems that sit on top of the canvas. Using these JavaScript libraries, you can pick the effects you want, without having to slog through all the math. The catch? There are at least a half-dozen high quality frameworks for canvas animation, each with its own distinct model and subtle quirks. And it's impossible to pick which of the evolving toolkits of today will become the best-supported and most widely accepted leaders of tomorrow. Some of the most worthy candidates today are Fabric.js (*http://fabricjs.com*), Paper.js (*http://paperjs.org*), EaselJS (*www.createjs.com*), and KineticJS (*http://kineticjs.com*). You can read some recent developer opinions on these libraries at *http://tinyurl.com/canvas-libraries*.

A Practical Example: The Maze Game

So far, you've explored how to combine the canvas with some key programming techniques to make interactive drawings and to perform animations. These building blocks take the canvas beyond mere drawing and into the realm of complete, self-contained applications—like games or Flash-style mini-apps.

Figure 9-11 shows a more ambitious example that takes advantage of what you've learned so far and builds on both concepts. It's a simple game that invites the user to guide a small happy face icon through a complex maze. When the user presses an arrow key, the happy face starts moving in that direction (using animation) and stops only when it hits a wall (using hit testing).

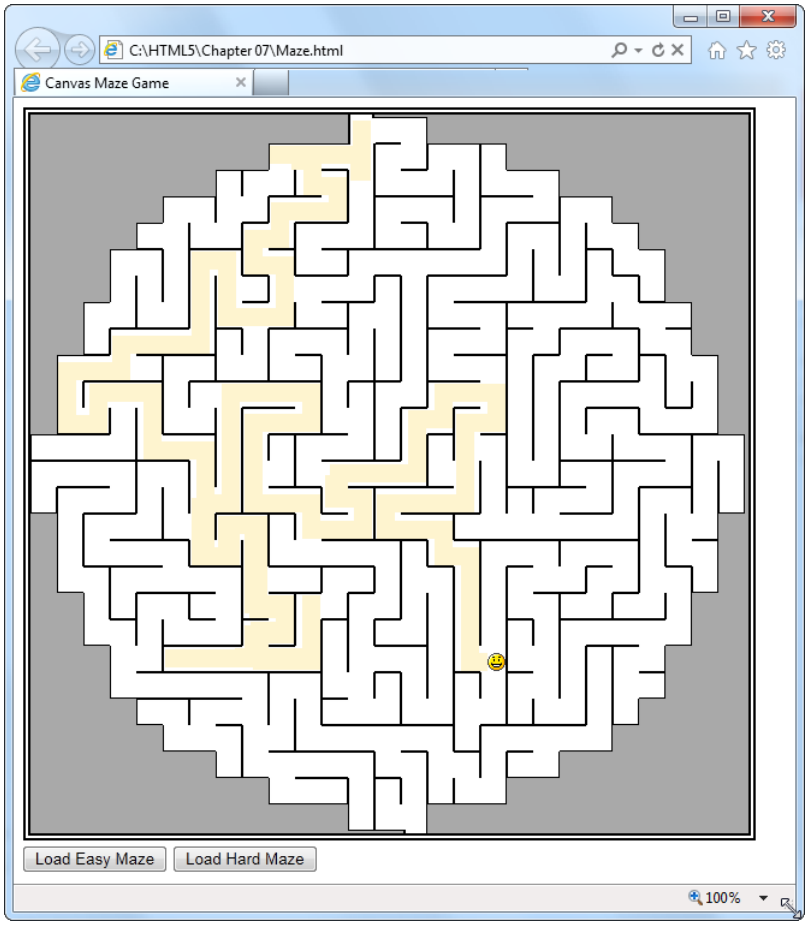

FIGURE 9-11
Guide the face through the maze. To a user, it looks like a fun game. To a developer, it's a potent mix of the HTML5 canvas and some smart JavaScript.

Of course, there's a trade-off. If you're going to rely on the canvas to build something sophisticated, you'll need to dig your way through a significant amount of code. The following sections show you the essentials, but be prepared to flex your JavaScript muscles.

NOTE You can run this example from your local computer if you're using Internet Explorer 9. But on other browsers, it works only if you first put your page (and the graphics it uses) on a test web server. To save the trouble, run this example from the try-out site at *http://prosetech.com/html5*.

Setting Up the Maze

Before anything can happen, the page needs to set up the canvas. Although you could paint the entire maze out of lines and rectangles, you'd need a lot of drawing code. Writing that drawing code by hand would be extremely tedious. You'd need to have a mental model of the entire maze, and then draw each short wall segment using a separate drawing operation. If you went this route, you'd almost certainly use a tool that automatically creates your drawing code. For example, you might draw the maze in Adobe Illustrator and then use a plug-in to convert it to canvas code (page 258).

Another option is to take a preexisting graphic of a maze and paint that on the canvas. This is particularly easy, because the web is filled with free pages that will create mazes for you. Using one of these pages, you set a few details—for example, the size, shape, colors, density, and complexity of the maze—and the page creates a downloadable graphic. (To try it out for yourself, just Google *maze generator*.)

This example uses maze pictures. When the page first loads, it takes a picture file (named *maze.png*) and draws that on the canvas. Here's the code that kicks that process off when the page first loads:

```
// Define the global variables for the canvas and drawing context.
var canvas;
var context;

window.onload = function() {
  // Set up the canvas.
  canvas = document.getElementById("canvas");
  context = canvas.getContext("2d");

  // Draw the maze background.
  drawMaze("maze.png", 268, 5);

  // When the user presses a key, run the processKey() function.
  window.onkeydown = processKey;
};
```

This code doesn't actually draw the maze background. Instead, it hands the work off to another function, named `drawMaze()`.

Because this example uses a separate maze-drawing function, it's not limited to a single maze. Instead, it lets you load any maze graphic you want. You simply need to call `drawMaze()` and pass in the file name of the maze picture, along with the coordinates for where the happy face should start. Here's the `drawMaze()` code that does the job:

```
// Keep track of the current face position.
var x = 0;
var y = 0;
```

```
function drawMaze(mazeFile, startingX, startingY) {
  // Load the maze picture.
  imgMaze = new Image();
  imgMaze.onload = function() {
    // Resize the canvas to match the size of the maze picture.
    canvas.width = imgMaze.width;
    canvas.height = imgMaze.height;

    // Draw the maze.
    var imgFace = document.getElementById("face");
    context.drawImage(imgMaze, 0,0);

    // Draw the face.
    x = startingX;
    y = startingY;

    context.drawImage(imgFace, x, y);
    context.stroke();

    // Draw the next frame in 10 milliseconds.
    setTimeout(drawFrame, 10);
  };
  imgMaze.src = mazeFile;
}
```

This code uses the two-step image drawing method explained on page 276. First, it sets a function that will handle the image's `onLoad` event and draw the maze image after it loads. Second, it sets the `src` attribute on the image object, which loads the image and triggers the code. This two-step process is a bit more complicated than just pulling the picture out of a hidden `<img>` element on the page, but it's necessary if you want to create a function that's flexible enough to load any maze picture you want.

When the maze image is loaded, the code adjusts the size of the canvas to match, places the face at its proper position, and then paints the face image. Finally, it calls `setTimeout()` to start drawing frames.

NOTE The downloadable version of this example (at *http://prosetech.com/html5*) is slightly more sophisticated. It lets the user load a new maze at any time—even while the happy face is still moving around the current maze. To make this work, it adds a bit of extra code in the `drawMaze()` function to stop the happy face (if it's currently moving) and halt the animating process, before loading the background and starting it again.

Animating the Face

When the user hits a key, the face begins moving. For example, if the user presses the down key, the happy face continues moving down until it hits a barrier or another key is pressed.

To make this work, the code uses two global variables that track the happy face's speed—in other words, how many pixels it will move in the *x* or *y* dimension, per frame. The variables that do this are named dx and dy, just as they were in the bouncing ball example (page 304). The difference is that you don't need an array for this page, because there's only one happy face:

```
var dx = 0;
var dy = 0;
```

When the user presses a key, the canvas calls the processKey() function. The function then checks if one of the arrow keys was pressed and adjusts the speed accordingly. To identify an arrow key, the code checks its key code against a known value. For example, a key code of 38 always represents the up arrow key. The processKey() function ignores all other keys except the arrow keys:

```
function processKey(e) {
  // If the face is moving, stop it.
  dx = 0;
  dy = 0;

  // The up arrow was pressed, so move up.
  if (e.keyCode == 38) {
    dy = -1;
  }

  // The down arrow was pressed, so move down.
  if (e.keyCode == 40) {
    dy = 1;
  }

  // The left arrow was pressed, so move left.
  if (e.keyCode == 37) {
    dx = -1;
  }

  // The right arrow was pressed, so move right.
  if (e.keyCode == 39) {
    dx = 1;
  }
}
```

The processKey() function doesn't change the current position of the happy face, or attempt to draw it. Instead, this task happens every 10 milliseconds, when the drawFrame() function is called.

The drawFrame() code is fairly straightforward, but detailed. It performs several tasks. First, it checks if the face is moving in either direction. If not, there's really no work to be done:

```
function drawFrame() {
  if (dx != 0 || dy != 0) {
```

If the face is moving, the drawFrame() code draws a yellow patch in the current face position (which creates the "trail" effect. It then moves the face to its new position:

```
    context.beginPath();
    context.fillStyle = "rgb(254,244,207)";
    context.rect(x, y, 15, 15);
    context.fill()

    // Increment the face's position.
    x += dx;
    y += dy;
```

Next, the code calls checkForCollision() to see if this new position is valid. (You'll see the code for this hit testing function in the next section.) If the new position isn't valid, the face has hit a wall, and the code must move the face back to its old position and stop it from moving:

```
    if (checkForCollision()) {
      x -= dx;
      y -= dy;
      dx = 0;
      dy = 0;
    }
```

Now the code is ready to draw the face, where it belongs:

```
    var imgFace = document.getElementById("face");
    context.drawImage(imgFace, x, y);
```

And check if the face has reached the bottom of the maze (and has thus completed it). If so, the code shows a message box:

```
    if (y > (canvas.height - 17)) {
      alert("You win!");
      return;
    }
  }
```

If not, the code sets a timeout so the drawFrame() method will be called again, 10 milliseconds later:

```
    // Draw a new frame in 10 milliseconds.
    setTimeout(drawFrame, 10);
}
```

You've now seen all the code for the maze game, except the innovative bit of logic in the `checkForCollision()` function, which handles the hit testing. That's the topic you'll tackle next.

Hit Testing with Pixel Colors

Earlier in this chapter, you saw how you can use mathematical calculations to do your hit testing. However, there's another approach you can use. Instead of looking through a collection of objects you've drawn, you can grab a block of pixels and look at their colors. This approach is simpler in some ways, because it doesn't need all the objects and the shape tracking code. But it works only if you can make clear-cut assumptions about the colors you're looking for.

NOTE The pixel-based hit-testing approach is the perfect approach for the maze example. Using this sort of hit testing, you can determine when the happy face runs into one of the black walls. Without this technique, you'd need a way to store all the maze information in memory and then determine if the current happy-face coordinates overlap one of the maze's wall lines.

The secret to the color-testing approach is the canvas's support for manipulating individual pixels—the tiny dots that comprise every picture. The drawing context provides three methods for managing pixels: `getImageData()`, `putImageData()`, and `createImageData()`. You use `getImageData()` to grab a block of pixels from a rectangular region and examine them (as in the maze game). You can also modify the pixels and use `putImageData()` to write them back to the canvas. And finally, `createImageData()` lets you create a new, empty block of pixels that exists only in memory, the idea being that you can customize them and then write them to the canvas with `putImageData()`.

To understand a bit more about the pixel-manipulation methods, consider the following code. First, it grabs a 100 x 50 square of pixels from the current canvas, using `getImageData()`:

```
// Get pixels starting at point (0,0), and stretching out 100 pixels to
// the right and 50 pixels down.
var imageData = context.getImageData(0, 0, 100, 50);
```

Then, the code retrieves the array of numbers that has the image data, using the `data` property:

```
var pixels = imageData.data;
```

You might expect that there's one number for each pixel, but life isn't that simple. There are actually *four* numbers for each pixel, one each to represent its red, green, blue, and alpha components. So if you want to examine each pixel, you need a loop that bounds through the array four steps at a time, like this:

```
// Loop over each pixel and invert the color.
for (var i = 0, n = pixels.length; i < n; i += 4) {

  // Get the data for one pixel.
  var red = pixels[i];
  var green = pixels[i+1];
  var blue = pixels[i+2];
  var alpha = pixels[i+3];

  // Invert the colors.
  pixels[i] = 255 - red;
  pixels[i+1] = 255 - green;
  pixels[i+2] = 255 - blue;
}
```

Each number ranges from 0 to 255. The code above uses one of the simplest image manipulation methods around—it inverts the colors. Try this with an ordinary picture, and you get a result that looks like a photo negative.

To see the results, you can write the pixels back to the canvas, in their original positions (although you could just as easily paint the content somewhere else):

```
context.putImageData(imageData, 0, 0);
```

The pixel-manipulation methods certainly give you a lot of control. However, they also have drawbacks. The pixel operations are slow, and the pixel data in the average canvas is immense. If you grab off a large chunk of picture data, you'll have tens of thousands of pixels to look at. And if you were already getting tired of drawing complex pictures using basic ingredients like lines and curves, you'll find that dealing with individual pixels is even more tedious.

That said, the pixel-manipulation methods can solve certain problems that would be difficult to deal with in any other way. For example, they provide the easiest way to create fractal patterns and Photoshop-style picture filters. In the maze game, they let you create a concise routine that checks the next move of the happy face icon and determines whether it collides with a wall. Here's the `checkForCollision()` function that handles the job:

```
function checkForCollision() {
  // Grab the block of pixels where the happy face is, but extend
  // the edges just a bit.
  var imgData = context.getImageData(x-1, y-1, 15+2, 15+2);
  var pixels = imgData.data;

  // Check these pixels.
  for (var i = 0; n = pixels.length, i < n; i += 4) {
    var red = pixels[i];
    var green = pixels[i+1];
    var blue = pixels[i+2];
```

```
      var alpha = pixels[i+3];

      // Look for black walls (which indicates a collision).
      if (red == 0 && green == 0 && blue == 0) {
        return true;
      }

      // Look for gray edge space (which indicates a collision).
      if (red == 169 && green == 169 && blue == 169) {
        return true;
      }
    }
    // There was no collision.
    return false;
  }
```

This completes the canvas maze game, which is the longest and most code-packed example you'll encounter in this book. It may take a bit more review (or a JavaScript brush-up) before you really feel comfortable with all the code it contains, but once you do you'll be able to use similar techniques in your own canvas creations.

If you're hungering for more canvas tutorials, you may want to check out a complete book on the subject, such as *HTML5 Canvas* (by Steve Fulton and Jeff Fulton) or *Core HTML5 Canvas* (by David Geary). Either book will take you deeper into the raw and gritty details of do-it-yourself canvas drawing.

POWER USERS' CLINIC

Eye-Popping Canvas Examples

There's virtually no limit to what you can do with the canvas. If you want to look at some even more ambitious examples that take HTML5 into the world of black-belt coding, the Web is your friend. Here's a list of websites that demonstrate some mind-blowing canvas mojo:

- **Canvas Demos.** This canvas example site has enough content to keep you mesmerized for days. Entries include the game Mutant Zombie Masters and the stock-charting tool TickerPlot. Visit *www.canvasdemos.com* to browse them all.
- **Wikipedia knowledge map.** This impressive canvas application shows a graphical representation of Wikipedia articles, with linked topics connected together by slender, web-like lines. Choose a new topic, and you zoom into that part of the knowledge map, pulling new articles to the forefront with a slick animation. See it at *http://en.inforapid.org*.
- **3D Walker.** This example lets you walk through a simple 3D world of walls and passages (similar to the ancient Wolfenstein 3D game that kicked off the first-person-shooter gaming craze way back in 1992). Take it for a spin at *www.benjoffe.com/code/demos/canvascape*.
- **Chess.** This HTML5 chess simulator lets you try your hand against a computer opponent with a canvas-drawn board that's rendered from above or with a three-dimensional perspective, depending on your preference. Challenge yourself to a game at *http://htmlchess.sourceforge.net/demo/example.html*.

PART 3

Building Web Apps

CHAPTER

10

Storing Your Data

On the Web, there are two places to store information: on the web server, or on the web client (the viewer's computer). Certain types of data belong on one, while others work better on the other.

The web server is the place to store sensitive information and data you don't want people tampering with. For example, if you fill your shopping cart at an online bookstore, your potential purchases are stored on the web server, as are the catalog of books, the history of past sales, and just about everything else. The only data your computer keeps is a tiny bit of tracking information that tells the website who you are, so it knows which shopping cart is yours. Even with HTML5, there's no reason to change this setup—it's safe, secure, and efficient.

But server-side storage isn't the best bet for every website. Sometimes, it's easier to keep nonessential information on the web surfer's computer. For example, local storage makes sense for *user preferences* (for example, settings that influence how the web page tailors its display) and *application state* (a snapshot of where the web application is right now, so the web visitor can pick up at the same spot later on). And if the data takes time to calculate or retrieve from the web server, you may be able to improve performance by storing it on the visitor's computer.

Before HTML5, the only way to get local storage was to use *cookies*, a mechanism originally devised to transmit small bits of identifying information between web browsers and web servers. Cookies work perfectly well for storing small amounts of data, but the JavaScript model for using them is a bit clunky. Cookies also force you to fiddle with expiry dates and needlessly send your data back and forth over the Internet with every web request.

HTML5 introduces a better alternative that lets you store information on your visitor's computer simply and easily. This data stays on the client indefinitely, isn't sent to the web server (unless you do it yourself), has plenty of room, and works through a couple of simple, streamlined JavaScript objects. This feature—called *web storage*—is a particularly nice fit with the offline application feature explored in Chapter 11, because it lets you build self-sufficient offline applications that can store all the information they need, even when there's no web connection.

In this chapter, you'll explore every corner of the web storage feature. You'll also look at two additional, newer standards: the File API, which lets the web browser read the content from other files on the computer's hard drive; and IndexedDB, which lets web developers run a complete, miniature database engine right inside the browser.

Web Storage Basics

HTML5's web storage feature lets a web page store some information on the viewer's computer. That information could be short-lived (so it disappears once the browser is shut down), or it could be long-lived (so it's still available days later, on subsequent visits to the website).

NOTE The name *web storage* is more than a little misleading. That's because the information a page stores is never on the Web—in fact, it never leaves the web surfer's computer.

There are two types of web storage, and they revolve around two objects:

- **Local storage** uses the `localStorage` object to store data permanently and make it accessible to any page in your website. If a web page stores local data, it will still be there when the visitor returns the next day, the next week, or the next year. Of course, most browsers also include a way to let users clear out local storage. Some web browsers provide an all-or-nothing command that lets people wipe out local data, in much the same way that you can clear out your cookies. (In fact, in some browsers the two features are linked, so that the only way to clear local data is to clear the cookies.) Other browsers may let their users review the storage usage of each website and clear the local data for specific sites.
- **Session storage** uses the `sessionStorage` object to store data temporarily, for a single window (or tab). The data remains until the visitor closes that tab, at which point the session ends and the data disappears. However, the session data stays around if the user goes to another website and then returns to your site, provided that this all happens in the same window tab.

TIP From the point of view of your web page code, both local storage and session storage work exactly the same. The difference is just how long the data lasts. Using local storage is the best bet for information you want to keep for future visits. Use session storage for data that you want to pass from one page to another. (You can also use session storage to keep temporary data that's used in just one page, but ordinary JavaScript variables work perfectly well for that purpose.)

Both local storage and session storage are linked to your website domain. So if you use local storage on a page at *www.GoatsCanFloat.org/game/zapper.html*, that data will be available on the page *www.GoatsCanFloat.org/contact.html*, because the domain is the same (*www.GoatsCanFloat.org*). However, other websites won't be able to see it or manipulate it.

Also, because web storage is stored on your computer (or mobile device), it's linked to that computer; a web page can't access information that was stored locally on someone else's computer. Similarly, you get different local storage if you log onto your computer with a different user name or fire up a different browser.

NOTE Although the HTML5 specification doesn't lay down any hard rules about maximum storage space, most browsers limit local storage to 5 MB. That's enough to pack in a lot of data, but it falls short if you want to use local storage to optimize performance by caching large pictures or videos (and truthfully, this isn't what local storage is designed to do). For space-hoggers, the still-evolving IndexedDB database storage standard (see page 340) offers much more room—typically hundreds of megabytes, if the user agrees.

Storing Data

To put a piece of information away into local storage or session storage, you first need to think of a descriptive name for it. This name is called a *key*, and you need it to retrieve your data later on.

To store a piece of data, you use the `localStorage.setItem()` method, as follows:

```
localStorage.setItem(keyName, data);
```

For example, imagine you want to store a piece of text with the current user's name. For this data, you might use the key name `user_name`, as shown here:

```
localStorage.setItem("user_name", "Marky Mark");
```

Of course, it doesn't really make sense to store a hard-coded piece of text. Instead, you'd store something that changes—for example, the current date, the result of a mathematical calculation, or some text that the user has typed into a text box. Here's an example of that last one:

```
// Get a text box.
var nameInput = document.getElementById("userName");

// Store the text from that text box.
localStorage.setItem("user_name", nameInput.value);
```

Pulling something out of local storage is as easy as putting it in, provided you use the `localStorage.getItem()` method. For example, here's a line of code that grabs the previously stored name and shows it in a message box:

```
alert("You stored: " + localStorage.getItem("user_name"));
```

This code works whether the name was stored five seconds ago or five months ago.

Of course, it's possible that nothing was stored at all. If you want to check whether a storage slot is empty, you can test for a null reference. Here's an example of that technique:

```
if (localStorage.getItem("user_name") == null) {
  alert ("You haven't entered a name yet.");
}
else {
  // Put the name into a text box.
  document.getElementById("userName").value = localStorage.getItem("user_
name");
}
```

Session storage is just as simple. The only difference is that you use the `sessionStorage` object instead of the `localStorage` object:

```
// Get the current date.
var today = new Date();

// Store the time as a piece of text in the form HH:mm.
var time = today.getHours() + ":" + today.getMinutes();
sessionStorage.setItem("lastUpdateTime", time);
```

Figure 10-1 shows a simple test page that puts all of these concepts together.

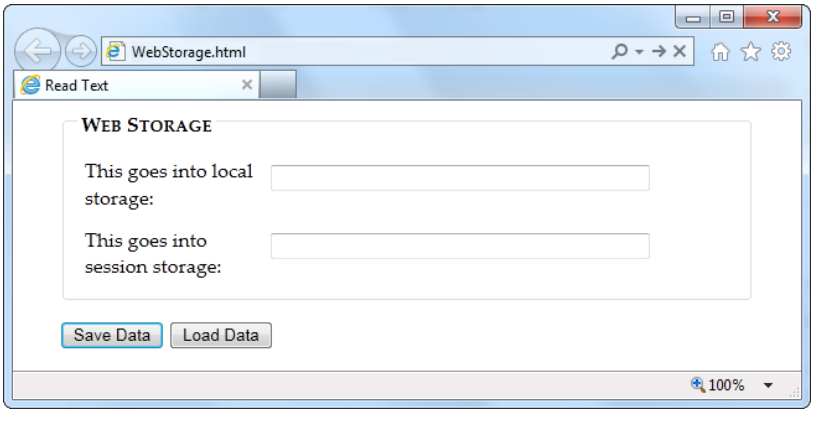

FIGURE 10-1

This page has two text boxes, one for session storage and one for local storage. When you click Save, the page stores the values. Click Load Data, and it brings them back. To try this out (and to verify that session storage disappears when the window is closed but that local storage lives forever), run this page at http://prosetech.com/html5.

NOTE Web storage also supports two alternate syntaxes for accessing data. Instead of using the `getItem()` and `setItem()` methods that you've already seen, you can use property names or an indexer. With property names, you access a storage slot called `"user_name"` as `localStorage.user_name`. With an indexer, you access the same storage slot as `localStorage["user_name"]`. You can choose a syntax based on your preference, but most web experts believe the `getItem()` and `setItem()` methods are best because they offer the least ambiguity.

TROUBLESHOOTING MOMENT

Web Storage Fails Without a Web Server

There's an unexpected problem that can trip up your web storage testing. In many browsers, web storage works only when you're requesting the pages from a live web server. It doesn't matter whether that web server is located on the Internet or if it's just a test server running on your own computer—the important detail is that you aren't just launching the pages from your local hard drive.

This quirk is a side effect of the way browsers dole out their local storage space. As you've already learned, they limit each website to 5 MB, and in order to do that, they need to associate every page that wants to use local storage to a website domain.

So what happens if you break this rule and open a web page that uses web storage, straight from a file? It depends. In Internet Explorer, the browser appears to lose its web storage support completely. The `localStorage` and `sessionStorage` objects disappear, and trying to use them causes a JavaScript error. In Firefox, the `localStorage` and `sessionStorage` objects remain, and support *appears* to be there (even to Modernizr), but everything you try to store quietly disappears into a void. And in Chrome, the result is different again—most of web storage works fine, but some features (like the `onStorage` event) don't work. You'll see the same issues when you use the File API (page 332). So do yourself a favor and put your pages on a test server, so you're not tripped up by unexpected quirks. Or, run the examples in this chapter from the try-out site at *http://prosetech.com/html5*.

A Practical Example: Storing the Last Position in a Game

At this point, you might have the impression that there isn't much to web storage, other than remembering to pick a name and put it in square brackets. And you'd be mostly right. But you can put local storage to some more practical purposes without any extra effort.

For example, consider the canvas-based maze game you saw in Chapter 9 (page 307). A maze might be too much trouble to solve in one go, in which case it makes sense to store the current position when the user closes the window or navigates to a new page. When the user returns to the maze page, your code can then restore the happy face to its previous position in the maze.

There are several possible ways to implement this example. You could simply save a new position after each move. Local storage is fast, so this wouldn't cause any problem. Or, you could react to the page's `onBeforeUnload` event to ask the game player whether it's worth storing the current position (Figure 10-2).

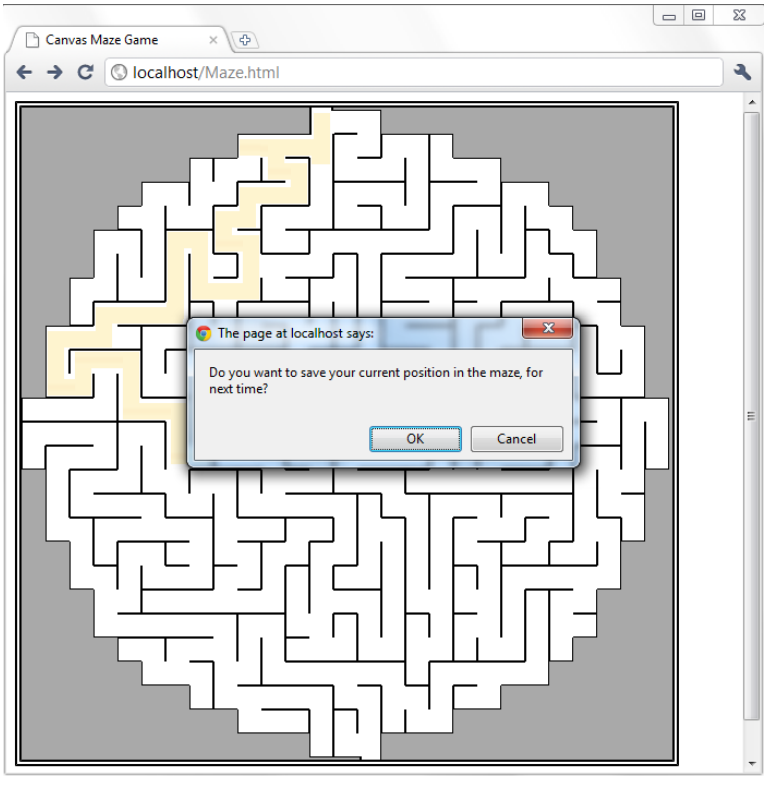

FIGURE 10-2

When a visitor leaves this page, either by navigating to a new site or by closing the window, the page offers to store the current position.

Here's the code that offers to store the position details:

```
window.onbeforeunload = function(e) {
  // Check if the localStorage object exists (as there's no reason to offer
  // to save the position if it won't work).
  if (localStorage) {

    // Ask to save the position.
    if (confirm(
  "Do you want to save your current position in the maze, for next time?")) {
      // Store the two coordinates in two storage slots.
      localStorage.setItem("mazeGame_currentX", x);
      localStorage.setItem("mazeGame_currentY", y);
    }
  }
}
```

TIP Long key names, like `mazeGame_currentX`, are good. After all, it's up to you to ensure that key names are unique, so two web pages on your website don't inadvertently use the same name to store different pieces of data. With just a single container that holds all your data, it's all too easy to run into naming conflicts, which is the one glaring weakness in the web storage system. To prevent problems, come up with a plan for creating logical, descriptive key names. For example, if you have separate maze games on separate pages, consider incorporating the page name into the key name, as in `Maze01_currentX`.

This example shows how to store an *application state* (the current position). If you wanted to avoid showing the same message each time the user leaves the game, you could add an "Automatically save position" checkbox. You would then store the position if the checkbox is switched on. Of course, you'd want to save the value of the checkbox too, and that would be an example of storing *application preferences*.

When the page loads the next time, you can check to see whether there's a previously stored position:

```
// Is the local storage feature supported?
if (localStorage) {
  // Try to get the data.
  var savedX = localStorage.getItem("mazeGame_currentX");
  var savedY = localStorage.getItem("mazeGame_currentY");

  // If the variables are null, no data was saved.
  // Otherwise, use the data to set new coordinates.
  if (savedX != null) x = Number(savedX);
  if (savedY != null) y = Number(savedY);
}
```

This example also uses the JavaScript `Number()` function to make sure the saved data is converted to valid numbers. You'll learn why that's important on page 327.

Browser Support for Web Storage

Web storage is one of the best-supported HTML5 features in modern browsers. The only browser that you're likely to find that doesn't support web storage is the thankfully endangered IE 7.

If you need a workaround for IE 7, you can simulate web storage using cookies. The fit isn't perfect, but it works. And although there's no official piece of script that plugs that gap, you can find many decent starting points on the GitHub polyfill page at *http://tinyurl.com/polyfill* (just look under the "Web Storage" section heading).

One web storage feature that enjoys slightly less support is the `onStorage` event, which you'll consider on page 330. In particular, IE 8 supports web storage but not the `onStorage` event. (IE 9 and later versions correct the problem, with full web storage support.) This situation is fine if you're using `onStorage` to add a nonessential feature, but otherwise be warned.

Deeper into Web Storage

You now know the essentials of web storage—how to put information in, and how to get it out again. However, there are several finer points and a few useful techniques left to cover before you put it to use. In the following sections, you'll see how to remove items from web storage and how to examine all the currently stored items. You'll also learn to deal with different data types, to store custom objects, and to react when the collection of stored items changes.

Removing Items

It couldn't be easier. You use the `removeItem()` method, and the key name, to get rid of a single piece of data you don't want:

```
localStorage.removeItem("user_name");
```

Or, if you want to empty out all the local data your website has stored, use the more radical `clear()` method:

```
sessionStorage.clear();
```

Finding All the Stored Items

To get a single piece of data out of web storage, you need to know its key name. But here's another neat trick. Using the `key()` method, you can pull every single item out of local or session storage (for the current website), even if you don't know any key names. This is a nifty technique when you're debugging, or if you just want to review what other pages in your site are storing, and what key names they're using.

Figure 10-3 shows a page that puts this technique into practice.

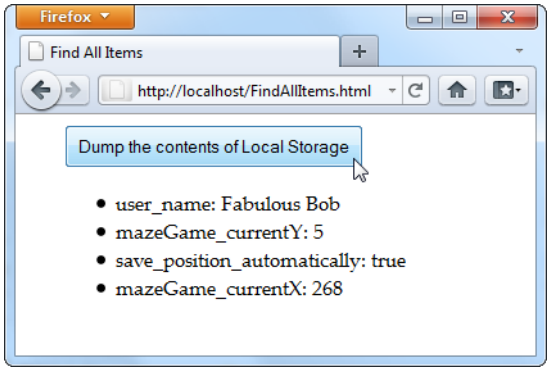

FIGURE 10-3

Click the button, and this page fills a list with the local storage contents.

When you click the button in this example, it triggers the `findAllItems()` function, which scans through the collection of items in local storage. Here's the code:

```
function findAllItems() {
  // Get the <ul> element for the list of items.
  var itemList = document.getElementById("itemList");

  // Clear the list.
  itemList.innerHTML = "";

  // Do a loop over all the items.
  for (var i=0; i<localStorage.length; i++) {
    // Get the key for the item at the current position.
    var key = localStorage.key(i);

    // Get the item that's stored with this key.
    var item = localStorage.getItem(key);

    // Create a new list item with this information,
    // and add it to the page.
    var newItem = document.createElement("li");
    newItem.innerHTML = key + ": " + item;
    itemList.appendChild(newItem);
  }
}
```

Storing Numbers and Dates

So far, the exploration into web storage has glossed over one important detail. Whenever you store something in the `localStorage` or `sessionStorage` object, that data is automatically converted to a string of text. For values that are already text (like the user name typed into a text box), there's no problem. But numbers aren't as forgiving. Consider the example on page 324, which stores the most recent maze position in local storage. If you forget to convert the coordinates from text to numbers, you can run into this sort of problem:

```
// Get the last x-coordinate position.
// For example, this might return the text "35"
x = localStorage.getItem("mazeGame_currentX");

// Attempt to increment the position (incorrectly).
x += 5;
```

Unfortunately, this code doesn't have the result you want. Because `x` is a string, JavaScript converts the number 5 to a string as well. And instead of adding `35+5`, JavaScript actually performs the string combination `"35"+"5"`, which returns a result of 355. This clearly isn't what you want. In fact, this code will cause the happy face to jump to completely the wrong position, or right out of the maze.

The issue here is that JavaScript assumes you're trying to stick two pieces of text together, rather than perform a mathematical operation. To solve the problem, you need to give JavaScript a hint that you're trying to work with numbers. Several solutions are possible, but the `Number()` function works well:

```
x = Number(localStorage.getItem("mazeGame_currentX"));

// Now JavaScript calculates 35+10 properly, and returns 40.
x += 10;
```

Text and numbers are easy to deal with, but if you want to place other sorts of data into web storage, you'll need to handle them with care. Some data types have handy conversion routines. For example, imagine you store a date like this:

```
var today = new Date();
```

This code doesn't store a date object, but a text string, like "Sat Jun 07 2014 13:30:46". Unfortunately, there's no easy way to convert this text back into a date object when you pull it out of storage. And if you don't have a date object, you won't be able to manipulate the date in the same way—say, calling date methods and performing date calculations.

To solve this problem, it's up to you to explicitly convert the date into the text you want, and then convert it back into a proper date object when you retrieve it. Here's an example:

```
// Create a date object.
var today = new Date();

// Turn the date into a text string in the standard form YYYY/MM/DD,
// and store that text.
var todayString = today.getFullYear() + "/" +
 today.getMonth() + "/" + today.getDate();
sessionStorage.setItem("session_started", todayString);

...

// Now retrieve the date text and use it to create a new date object.
// This works because the date text is in a recognizable format.
today = new Date(sessionStorage.getItem("session_started"));

// Use the methods of the date object, like getFullYear().
alert(today.getFullYear());
```

Run this code, and the year appears in a message box, confirming that you've successfully recreated the date object.

Storing Objects

In the previous section, you saw how to convert numbers and dates to text and back again, so you can store them with web storage. These examples work because the JavaScript language helps you out, first with the handy `Number()` function, and then with the text-to-date conversion smarts that are hard-wired into date objects. However, there are plenty of other objects that you can't convert this way, especially if you create a custom object of your own.

For example, consider the personality quiz you first saw in Chapter 9 (page 290). The personality quiz uses two pages. On the first, the quiz-taker answers some questions and gets a score. On the second, the results are shown. In the original version of this page, the information is passed from the first page to the second using query string arguments that are embedded in the URL. This approach is traditional HTML (although a cookie would work too). But in HTML5, local storage is the best way to shuffle data around.

But here's the challenge. The quiz data consists of five numbers, one for each personality factor. You *could* store each personality factor in a separate storage slot. But wouldn't it be neater and cleaner to create a custom object that packages up all the personality information in one place? Here's an example of a `PersonalityScore` object that does the trick:

```
function PersonalityScore(o, c, e, a, n) {
  this.openness = o;
  this.conscientiousness = c;
  this.extraversion = e;
  this.accommodation = a;
  this.neuroticism = n;
}
```

If you create a `PersonalityScore` object, you need just one storage slot, instead of five. (For a refresher about how custom objects work in JavaScript, see page 468.)

To store a custom object in web storage, you need a way to convert the object to a text representation. You could write some tedious code that does the work. But fortunately, there's a simpler, standardized approach called *JSON encoding*.

JSON (JavaScript Object Notation) is a lightweight format that translates structured data—like all the values that are wrapped in an object—into text. The best thing about JSON is that browsers support it natively. That means you can call `JSON.stringify()` to turn any JavaScript object into text, complete with all its data, and `JSON.parse()` to convert that text back into an object. Here's an example that puts this to the test with the `PersonalityScore` object. When the test is submitted, the page calculates the score (not shown), creates the object, stores it, and then redirects to the new page:

```
// Create the PersonalityScore object.
var score = new PersonalityScore(o, c, e, a, n);
```

```
// Store it, in handy JSON format.
sessionStorage.setItem("personalityScore", JSON.stringify(score));

// Go to the results page.
window.location = "PersonalityTest_Score.html";
```

On the new page, you can pull the JSON text out of storage and use the `JSON.parse()` method to convert it back to the object you want. Here's that step:

```
// Convert the JSON text to a proper object.
var score = JSON.parse(sessionStorage.getItem("personalityScore"));

// Get some data out of the object.
lblScoreInfo.innerHTML = "Your extraversion score is " + score.extraversion;
```

To see the complete code for this example, including the calculations for each personality factor, visit *http://prosetech.com/html5*. To learn more about JSON and take a peek at what JSON-encoded data actually looks like, check out *http://en.wikipedia.org/wiki/JSON*.

Reacting to Storage Changes

Web storage also gives you a way to communicate among different browser windows. That's because whenever a change happens to local storage or session storage, the `window.onStorage` event is triggered in every other window that's viewing the same page or another page on the same website. So if you change local storage on *www.GoatsCanFloat.org/storeStuff.html*, the `onStorage` event will fire in a browser window for the page *www.GoatsCanFloat.org/checkStorage.html*. (Of course, the page has to be viewed in the same browser and on the same computer, but you already knew that.)

The `onStorage` event is triggered whenever you add a new object to storage, change an object, remove an object, or clear the entire collection. It doesn't happen if your code makes a storage operation that has no effect (like storing the same value that's already stored, or clearing an already-empty storage collection).

Consider the test page shown in Figure 10-4. Here, the visitor can add any value to local storage, with any key, just by filling out two text boxes. When a change is made, the second page reports the new value.

To create the example shown in Figure 10-4, you first need to create the page that stores the data. In this case, clicking the Add button triggers a short `addValue()` function that looks like this:

```
function addValue() {
  // Get the values from both text boxes.
  var key = document.getElementById("key").value;
  var item = document.getElementById("item").value;
```

```
  // Put the item in local storage.
  // (If the key already exists, the new item replaces the old.)
  localStorage.setItem(key, item);
}
```

The second page is just as simple. When the page first loads, it attaches a function to the `window.onStorage` event, using this code:

```
window.onload = function() {
  // Connect the onStorage event to the storageChanged() function.
  window.addEventListener("storage", storageChanged, false);
};
```

This code looks a little different than the event handling code you've seen so far. Instead of setting `window.onstorage`, it calls `window.addEventListener()`. That's because this code is the simplest that works on all current browsers. If you set `window.onstorage` directly, your code will work in every browser except Firefox.

NOTE Web graybeards may remember that the `addEventListener()` method doesn't work on Internet Explorer 8 (or older). In this example, that limitation is no cause for concern, because IE 8 doesn't support storage events anyway.

The `storageChanged()` function has a simple task. It grabs the updated information and displays it on the page, in a `<div>` element:

```
function storageChanged(e) {
  var message = document.getElementById("updateMessage");
  message.innerHTML = "Local storage updated.";
  message.innerHTML += "<br>Key: " + e.key;
  message.innerHTML += "<br>Old Value: " + e.oldValue;
  message.innerHTML += "<br>New Value: " + e.newValue;
  message.innerHTML += "<br>URL: " + e.url;
}
```

As you can see, the `onStorage` event provides several pieces of information, including the key of the value that was changed, the old value, the newly applied value, and the URL of the page that made the change. If the `onStorage` event is a reaction to the insertion of a new item, the `e.oldValue` property is either null (in most browsers) or an empty string (in Internet Explorer).

NOTE If you have several pages open for the same website, the `onStorage` event occurs in each one, except the page that made the change (in the current example, that's *StorageEvents1.html*). However, Internet Explorer is the exception—it doesn't follow this rule, and fires the `onStorage` event in the original page as well.

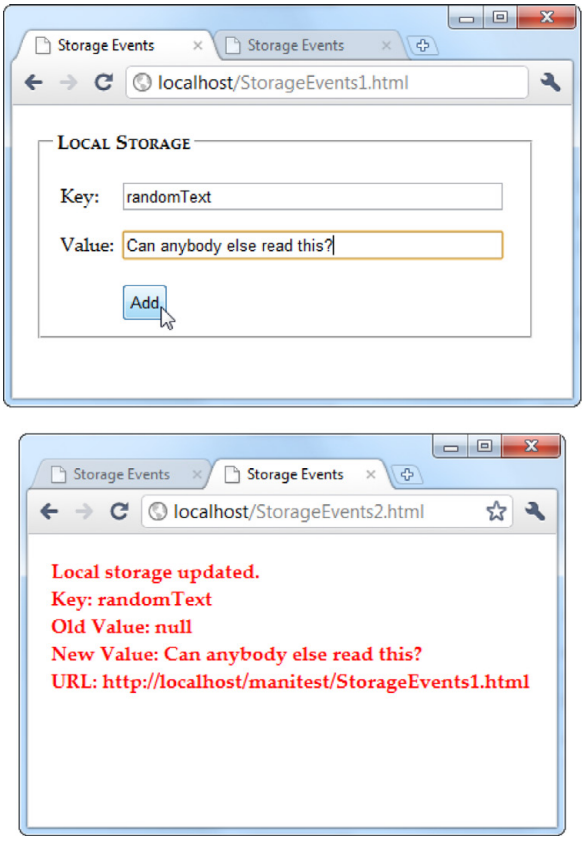

FIGURE 10-4

To see the onStorage *event in action, open* Storage Events1.html *and* StorageEvents2.html *at the same time. When you add or change a value in the first page (top), the second page captures the event and reports it in the page (bottom).*

Reading Files

Web storage is a solidly supported part of HTML5. But it's not the only way to access information. Several new standards are creeping onto the field for different types of storage-related tasks. One example is a standard called the File API, which technically isn't a part of HTML5, but has good support across modern browsers, with the exception of Internet Explorer (which didn't add support until IE 10).

Based on its rather vague name, you might expect that the File API is a sweeping standard for reading and writing files on a web browser's hard drive. However, it's not that ambitious or powerful. Instead, the File API gives a way for a visitor to pick a file from his hard drive and hand it directly to the JavaScript code that's running

on the web page. This code can then open the file and explore its data, whether it's simple text or something more complicated. The key thing is that the file goes directly to the JavaScript code. Unlike a normal file upload, it never needs to travel to the web server.

It's also important to note what the File API *can't* do. Most significantly, it can't change a file or create a new one. If you want to store any data, you'll need to use another mechanism—for example, you can send your data to a web server through `XMLHttpRequest` (page 377), or you can put it in local storage.

You might think that the File API is less useful than local storage—and for most websites, you'd be right. However, the File API is also a toehold into a world where HTML has never gone before, at least not without a plug-in to help.

NOTE Right now, the File API is an indispensable feature for certain types of specialized applications, but in the future its capabilities may expand to make it much more important. For example, future versions may allow web pages to *write* local files, provided the user has control over the file name and location, using a Save dialog box. Browser plug-ins like Flash already have this capability.

Getting Hold of a File

Before you can do anything with the File API, you need to get hold of a file. There are three strategies you can use, but they are the same in one key fact—namely, your web page gets a file only if the visitor explicitly picks it and gives it to you.

Here are your options:

- **The `<input>` element.** Set the type attribute to file, and you've got the standard file upload box. But with a bit of JavaScript and the File API, you can open it locally.
- **A hidden `<input>` element.** The `<input>` element is ugly. To get right with the style police, you can hide your `<input>` element and make a nicer browser button. When it's clicked, use JavaScript to call the `click()` method on the hidden `<input>` element. This shows the standard file-selection dialog box.
- **Drag-and-drop.** If the browser supports it, you can drag a file from the desktop or a file browser window and drop it on a region in the web page.

In the following sections, you'll see all three approaches.

Reading a Text File with <input>

One of the easiest things you can do with the File API is read the content from a simple text file. Figure 10-5 shows a web page that uses this technique to read the markup in a web page and then display it.

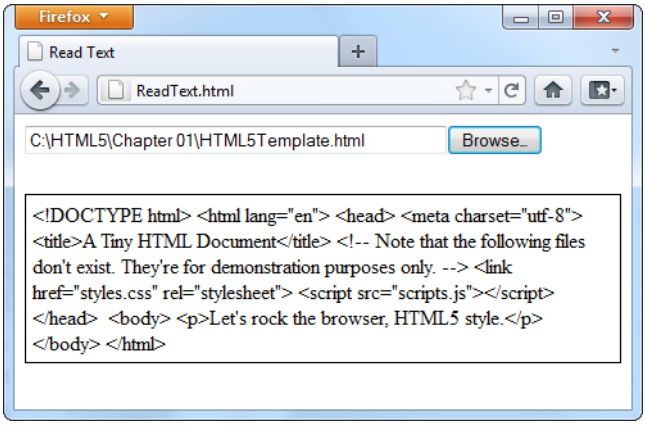

FIGURE 10-5

Click the Browse button (or Choose File, as it's named in Chrome), choose a file, and then click OK. Instead of the usual upload, the web page JavaScript takes it from there, copying the content into your page.

To build this example, you start with the `<input type="file">` element, which creates the infamous text box and Browse button combination:

```
<input id="fileInput" type="file" onchange="processFiles(this.files)">
```

However, whereas an `<input>` element usually belongs in a `<form>` container so the file can be posted to the web server, this `<input>` element goes its own way. When the web page visitor picks a file, it triggers the `<input>` element's onChange event, and that triggers the `processFiles()` function. It's here that the file is opened, in ordinary JavaScript.

Now you'll consider the `processFiles()` function, one piece at a time. The first order of business is to grab the first file from the collection of files that the `<input>` element provides. Unless you explicitly allow multiple file selection (with the `multiple` attribute), the files collection is guaranteed to hold just one file, which will be at position 0 in the files array:

```
function processFiles(files) {
  var file = files[0];
```

NOTE Every file object has three potentially useful properties. The name property gives you its file name (not including the path), the size property tells you how many bytes big it is, and the type property tells you the MIME type of the file (page 152), if it can be determined. You could read these properties and add additional logic—for example, you could refuse to process files above a certain size, or allow files of only a certain type.

Next, you create a `FileReader` object that allows you to process the file:

```
var reader = new FileReader();
```

It's almost time to call one of the FileReader's methods to extract the file content. All of these methods are *asynchronous*, which means they start the file-reading task but don't wait for the data. To get the data, you first need to handle the onLoad event:

```
reader.onload = function (e) {
  // When this event fires, the data is ready.
  // Copy it to a <div> on the page.
  var output = document.getElementById("fileOutput");
  output.textContent = e.target.result;
};
```

Finally, with that event handler in place, you can call the FileReader's readAsText() method:

```
  reader.readAsText(file);
}
```

This method dumps all the file content into a single long string, which is provided in the e.target.result that's sent to the onLoad event.

The readAsText() method works properly only if the file holds text content (not binary content). That means it suits HTML files perfectly well, as shown in Figure 10-5. However, there are other useful formats that use plain text. One good example is the CSV format, which is a basic export format supported by all spreadsheet programs. Another example is the XML format, which is a standard way of exchanging bunches of data between programs. (It's also the foundation of the Office XML formats, which means you can hand .docx and .xlsx files directly to the readAsText() method as well.)

NOTE The JavaScript language even has a built-in XML parser, which means you can browse around an XML file and dig out just the content you need. Of course, the code this requires is considerable, it performs poorly for large files, and it's rarely easier than just uploading the file to a web server and running your file-processing logic there. However, you can start to see how the File API can open up new possibilities that HTML lovers didn't dare imagine even just a few years ago.

The readAsText() method isn't the only way to pull your data out of a file. The FileReader object also includes the following file-reading methods: readAsBinaryString(), readAsArrayBuffer(), and readAsDataUrl().

The readAsBinaryString() method gives your application the ability to deal with binary-encoded data, although it somewhat awkwardly sticks it into a text string, which is inefficient. And if you actually want to *interpret* that data, you need to struggle through some horribly convoluted code.

The readAsArryaBuffer() method is a better bet for web developers who need to do some serious do-it-yourself data processing. This method reads the data into an array (page 465), where each element represents a single byte of data. The advantage of this package is that you can use it to create Blob objects and slice out smaller sections of binary data, so you can process it one chunk at a time.

NOTE Blob is shorthand for *binary large object*—in other words, a fat chunk of raw data. Blobs are an advanced part of the File API; to learn more, you can take a look at Mozilla's steadily evolving documentation on the subject at *http://tinyurl.com/file-blob*.

Lastly, the `readAsDataURL()` method gives you an easy way to grab picture data. You'll use that on page 339. But first, it's time to make the page in this example a bit prettier.

Replacing the Standard Upload Control

Web developers agree: The standard `<input>` control for file submission is ugly. And although you do need to use it, you don't need to let anyone see it. Instead, you can simply hide it, with a style rule like this:

```
#fileInput {
  display: none;
}
```

Now add a new control that will trigger the file-submission process. An ordinary link button will do, which you can make look as pretty as you want:

```
<button onclick="showFileInput()">Analyze a File</button>
```

The final step is to handle the button click and use it to manually trigger the `<input>` element, by calling *its* `click()` method:

```
function showFileInput() {
  var fileInput = document.getElementById("fileInput");
  fileInput.click();
}
```

Now, when the button is clicked, the `showFileInput()` function runs, which clicks the hidden Browse button and shows the dialog box where the visitor can pick a file. This, in turn, triggers the hidden `<input>` element's `onChange` event, which runs the `processFiles()` function, just like before.

Reading Multiple Files at Once

There's no reason to limit people to submitting one file at a time. HTML5 allows multiple file submission, but you need to explicitly allow it by adding the `multiple` attribute to the `<input>` element:

```
<input id="fileInput" type="file" onchange="processFiles(this.files)"
multiple>
```

Now the user can select more than one file in the Open dialog box (for example, by pressing Ctrl on a Windows computer while clicking several files, or by dragging a box around a group of them). Once you allow multiple files, your code needs to accommodate them. That means you can't just grab the first file from the files collection, as in the previous example. Instead, you need a `for` loop that processes each file, one at a time:

```
for (var i=0; i<files.length; i++) {
  // Get the next file
  var file = files[i];

  // Create a FileReader for this file, and run the usual code here.
  var reader = new FileReader();
  reader.onload = function (e) {
    ...
  };
  reader.readAsText(file);
}
```

Reading an Image File with Drag-and-Drop

As you've seen, `FileReader` handles text content in a single, simple step. It deals will images just as easily, thanks to the `readAsDataURL()` method.

Figure 10-6 shows an example that introduces two new features: image support, and file drag-and-drop. The submitted image is used to paint the background of an element, although you could just as easily paint it on a canvas and process it using the canvas's raw pixel features (page 313). For example, you could use this technique to create a page where someone can drop in a picture, draw on it or modify it, and then upload the final result using an `XMLHttpRequest` call (page 377).

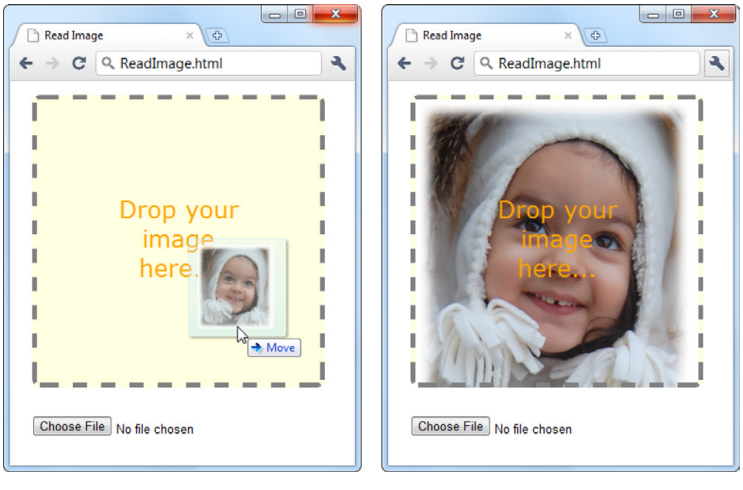

FIGURE 10-6

There are two ways to provide an image to this page: Use the file controls below, or drag one or more images onto the drop box.

To create this page, you first need to decide what element will capture the dropped files. In this example, it's a `<div>` element named `dropBox`:

```
<div id="dropBox">
  <div>Drop your image here...</div>
</div>
```

Some straightforward style sheet magic gives the drop box the size, borders, and colors you want:

```
#dropBox {
  margin: 15px;
  width: 300px;
  height: 300px;
  border: 5px dashed gray;
  border-radius: 8px;
  background: lightyellow;
  background-size: 100%;
  background-repeat: no-repeat;
  text-align: center;
}

#dropBox div {
  margin: 100px 70px;
  color: orange;
  font-size: 25px;
  font-family: Verdana, Arial, sans-serif;
}
```

Keen eyes will notice that the drop box sets the `background-size` and `background-repeat` properties in preparation for what comes next. When the image is dropped onto the `<div>`, it's set as the background. The `background-size` property ensures that the picture is compressed down so you can see all of it at once. The `background-repeat` property ensures that the picture isn't tiled to fill the leftover space.

To handle file drops, you need to handle three events: `onDragEnter`, `onDragOver`, and `onDrop`. When this page first loads, it attaches event handlers for all three:

```
var dropBox;

window.onload = function() {
  dropBox = document.getElementById("dropBox");
  dropBox.ondragenter = ignoreDrag;
  dropBox.ondragover = ignoreDrag;
  dropBox.ondrop = drop;
};
```

The `ignoreDrag()` function handles both `onDragEnter` (which fires when the mouse pointer enters the drop box, with a file attached) and `onDragOver` (which fires continuously, as the file-dragging mouse pointer moves over the drop box). That's because you don't need to react to either of these actions, other than to tell the browser to hold off from taking any actions of its own. Here's the code you need:

```
function ignoreDrag(e) {
  // Make sure nobody else gets this event, because you're handling
  // the drag and drop.
```

```
    e.stopPropagation();
    e.preventDefault();
}
```

The onDrop event is more important. It's at this point that you get the file and process it. However, because there are actually two ways to submit files to this page, the drop() function calls the processFiles() function to do the actual work:

```
function drop(e) {
  // Cancel this event for everyone else.
  e.stopPropagation();
  e.preventDefault();

  // Get the dragged-in files.
  var data = e.dataTransfer;
  var files = data.files;

  // Pass them to the file-processing function.
  processFiles(files);
}
```

The processFiles() function is the last stop in the drag-and-drop journey. It creates a FileReader, attaches a function to the onload event, and calls readAsDataURL() to transform the image data into a data URL (page 269):

NOTE As you learned when you explored the canvas, a data URL is a way of representing image data in a long text string that's fit for URL. This gives you a portable way to move the image data around. To show the image content in a web page, you can set the src property of an <img> element (as on page 276), or you can set the CSS background-image property (as in this example).

```
function processFiles(files) {
  var file = files[0];

  // Create the FileReader.
  var reader = new FileReader();

  // Tell it what to do when the data URL is ready.
  reader.onload = function (e) {
    // Use the image URL to paint the drop box background
    dropBox.style.backgroundImage = "url('" + e.target.result + "')";
  };

  // Start reading the image.
  reader.readAsDataURL(file);
}
```

The FileReader provides several more events, and when reading image files you might choose to use them. The onProgress event fires periodically during long operations, to let you know how much data has been loaded so far. (You can cancel an operation that's not yet complete by calling the FileReader's abort() method.) The onError event fires if there was a problem opening or reading the file. And the onLoadEnd event fires when the operation is complete for any reason, including if an error forced it to end early.

Browser Support for the File API

The File API has solid browser support, but it isn't quite as reliable as web storage. Table 10-2 shows which browsers include it.

TABLE 10-1 *Browser support for the File API*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	ANDROID
Minimum version	10	3.6	13	6	11.1	6	3

Because the File API requires some privileges that ordinary web pages don't have, you can't patch the missing feature with more JavaScript. Instead, you need to rely on a plug-in like Flash or Silverlight. For example, you can find a polyfill at *https://github.com/MrSwitch/dropfile* that uses Silverlight to intercept a dragged file, open it, and then pass the details to the JavaScript code on the page.

IndexedDB: A Database Engine in a Browser

You're probably familiar with the concept of *databases*—carefully structured catalogs of information that can swallow lists of people, products, sales, and just about any other sort of data you want to stuff them with. Many websites use databases that are stored on the web server. For example, when you look up a book on Amazon, the page extracts the details from Amazon's staggeringly big database. Much the same thing happens when you look for an article on Wikipedia or a video on YouTube, or when you perform a web search on Google.

For years, the story ended there. Databases were the province of the web server; conversation closed. But HTML5 introduces another possibility. What if the browser had the ability to create a *local* database—a database that's stored on the client's computer rather than the far-off web server? Different ideas cropped up, including a briefly popular and now abandoned specification called Web SQL Databases. More recently, the still-evolving IndexedDB specification rose to prominence as the official HTML5-sanctioned solution for local databases.

NOTE The name "IndexedDB" emphasizes the fact that the databases use *indexes* to identify and arrange data. Indexes ensure that you don't end up with the same record of data stored in two places, and they also speed up data retrieval when you have acres of information.

UP TO SPEED

The Difference Between Server-Side and Client-Side Databases

There are plenty of ways for websites to use server-side databases, but none of them have anything to do with HTML5. To understand why, you need to remember that the HTML5 universe is centered on the mix of markup, styles, and JavaScript that's loaded in the client's browser. There's no direct way for a browser on your computer to access a database on the web server. Even if it were technically possible, this kind of contact would raise all kinds of security issues. Instead, to interact with a server-side database you need code that runs on the web server. For example, websites commonly use PHP or C# code (although there are many other technologies) to take care of server-side database tasks like generating a page of search results or logging a customer's purchase.

Before going ahead, it's important to understand exactly what role IndexedDB can play in a web application. For a host of reasons, IndexedDB can never replace the server-side databases described earlier. Most significantly, there's this: IndexedDB creates a separate database for every user. Websites that use server-side data need a single, centralized catalog of information that they can share with everyone.

However, IndexedDB is useful in several other special scenarios:

- **Making a self-sufficient offline application.** In Chapter 11, you'll learn how HTML5 caching lets you run a web page even without a web connection—a feat that's particularly useful for mobile devices. You can make your offline apps even more capable by using IndexedDB storage. For example, your page can retrieve the data it needs from a database on the web server when it's connected (using the `XMLHttpRequest` object demonstrated on page 377), and store that data in a local database so it's available even when the network isn't.
- **Enhancing performance.** Some applications use masses of data, and if you're continually retrieving the same data every time you need it, you'll slow down your pages. Similarly, if you create an application that generates data using complex calculations, you don't want to waste time performing the same work more than once. The solution is to store everything you need in an IndexedDB database. Treat it like your own super-customizable cache.
- **Improving local storage.** As you learned earlier, local storage provides a place to stash some data between browser sessions, and pass it from one page to another. If your data is relatively simple in structure and small in size, ordinary JavaScript variables and objects will hold it fine. But if you have an extraordinarily large or complex batch of data, you may find that it's easier and faster to manage it with an IndexedDB database.

NOTE The IndexedDB storage system behaves like local storage in several key ways. Most importantly, an IndexedDB database belongs to a single person, using a particular browser and a particular computer, and visiting a specific site. If you change any of these details—for example, switching browsers, logging on with a different user account, or switching to your smartphone—the web page gets a new IndexedDB database.

Learning to use IndexedDB can be daunting. First, the specification is complex (some less charitable developers say ugly). You, the web developer, are responsible for creating the database, building its tables, and filling them with data. Second, IndexedDB uses an *asynchronous model*, which means that database tasks happen in the background, without stalling your code or locking up the page. The drawback is that this forces you to use a more complex model that scatters your code into different places. For example, the code you use to start a database task isn't in the same place as the code you use to handle the outcome of that task. Following the sequence and figuring out how each piece of code fits together takes some practice.

Figure 10-7 shows the database-powered web page that you'll analyze through the rest of this chapter. It puts the IndexedDB feature through a set of basic operations with a database that stores link information. Each record in the database consists of a URL and some relevant information about that URL, like its name and a description. Using the *FavoriteSiteTracker.html* page, you can add new link records, browse the ones that already exist, and change or delete individual records—all of which are fundamental tasks for code that deals with any type of database.

The Data Object

Before you get started with the link tracker example, it's time to learn about the data it stores—the `LinkRecord` object. Traditional databases are all about tables and fields, but IndexedDB simplifies life by storing *objects*. So before you create your database, it's a good idea to define your data structure in JavaScript.

In the link tracker example, the database has a single table, and each record in that table holds the information for a single link. You can package all the link details in JavaScript using an object-definition function, like this:

```
function LinkRecord(name, url, description, notes) {
  this.name = name;
  this.url = url;
  this.description = description;
  this.notes = notes;
}
```

Once you've added this function to your code, you can call it when needed to create a `LinkRecord` object, complete with a descriptive name, URL, full description, and notes. (If you're a bit fuzzy on object-creation with JavaScript, refer to page 468 for a review.)

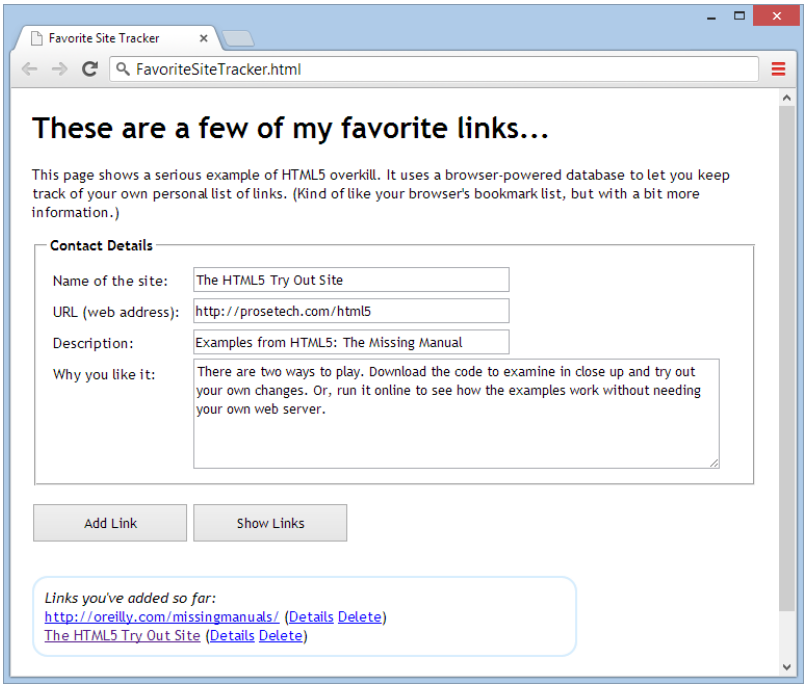

FIGURE 10-7

This example uses an IndexedDB database to hold link information. The top of the page has a form that lets you create a new link record. Underneath is a list that shows the name and URL (but not the other details) of every link that's in the database. As always, you can try this example for yourself at http://prosetech.com/html5.

Creating and Connecting to a Database

In any web page that uses IndexedDB, the first task is to connect to your database. Since you'll usually perform this task when the page first loads, it makes sense to handle the `window.onload` event and put your code there. Here's the basic skeleton:

```
var database;

window.onload = function () {
  ...
}
```

The `database` variable goes outside your function. That way, once you set the database variable, you can access it everywhere else in your code.

So, what does your code need to do when the page loads? That depends on what you've done before:

- If this is the first time the user is visiting the page, you need to create the database and set up its tables from scratch.
- If the user has been here before, you only need to open the database and set the database variable, so it's ready for use.

Either way, your starting point is the `open()` function that's provided by the `window.indexedDB` object. It takes two arguments: the name of the database you want to use (or create), and its version number. If you're starting out with a brand new database, you need to set the version to 1:

```
var request = window.indexedDB.open("LinksDB", 1);
```

NOTE It doesn't matter what name you give your database, so long as you stay consistent and aren't already using that name in another page on the same website.

As explained earlier, everything in IndexedDB land is asynchronous. Accordingly, when you call the `open()` method, nothing is actually opened. Instead, you're making a *request*. In this case, your request is for the browser to open the database when it gets the chance, and preferably on a background thread so it won't freeze up the page for even a moment.

You'll find this pattern of asynchronous requests—and the same request object—with just about every task you perform with IndexedDB. To react when the task is finished, you need to attach at least two event handlers: one that handles a successful result (`onSuccess`) and one that handles an unexpected failure (`onError`). In addition, when using the `open()` method, you need a third event handler for an event called `onUpgradeNeeded`.

When opening a database, the `onSuccess` event indicates that the right version of your database already exists. At this point you need to set the database variable so you can use it later. Optionally, you can get to work reading data. In the link tracker example, a separate function named `showLinks()` does that job (page 349).

```
request.onsuccess = function(event) {
  alert("Created or opened database successfully.");

  // Make the database available everywhere in the code.
  database = request.result;

  // Call the showLinks() function to read the database and display the
  // list of links that already exist.
  showLinks();
};
```

The onError event is equally straightforward. Depending on the error, you may be able to take corrective action, or you may simply choose to report it to the user with a message either in a pop-up box or right on the page.

```
request.onerror = function(event) {
  alert(request.error + " occurred.");
};
```

WARNING Don't make the mistake of omitting the onError event handler. If you do, problems will slip by without any notification at all, and you'll have no clue what went wrong.

The onUpgradeNeeded event is the most interesting of the three. It fires when the database version you requested isn't available. For example, if you requested database version 2 and the computer currently has database version 1, this event fires. At this point, you can upgrade the database—for example, by adding a new table. (You can find out the version of the current database from the event.oldVersion argument.) The onUpgradeNeeded event also fires if the database doesn't exist at all, in which case the browser obligingly creates a blank database and waits for you to fill in the tables you want.

To create a table, you must first grab the database object from the request.result property. Then, you call its createObjectStore() method. Here's the code that adds a table named Links to the LinksDB database:

```
  request.onupgradeneeded = function(event) {
    var db = request.result;
    var objectStore = db.createObjectStore("Links", { keyPath: "url" });
  }
};
```

In traditional database programming, you need to specify field names and data types when you create a table. But IndexedDB uses a simpler, object-focused model. When you create a table, you supply just two details: the name of the table and the *key path*, which is the property of your object that should be used for the primary key.

NOTE The key path identifies a property in your data object. In this example, the word *url* is written in lowercase so it matches the name of the property of the data object. If you flip back to page 342, you'll see that the LinkRecord object defines a property named url.

UP TO SPEED

Understanding Primary Keys

The *primary key* is the bit of information that uniquely identifies each record. In the case of the Links table, each link record has a unique URL, so it makes sense to use that for the key. This setup prevents you from accidentally creating two records for the same address. It also forces the browser to create an index for the Links table that will let you look up any link record by URL.

If your data doesn't include an obvious candidate for the primary key—in other words, it doesn't have an essential piece of information that's guaranteed to be unique—you can use the time-honored database technique of adding a numeric ID number. Even better, you can get the browser to generate unique numbers for you automatically and fill them in whenever you insert a new record. To do that, set the `autoIncrement` property to `true` when you create the database:

```
var objectStore =
db.createObjectStore("Links",
 { keyPath: "id", autoIncrement: true });
```

If your database needs to have multiple tables, you can call `createObjectStore()` multiple times, with different data objects. The `request.onError` event fires if the browser encounters a problem. Once the tables are added successfully, the `request.onSuccess` event will fire and your code can finish setting up the page.

NOTE The databases you create with IndexedDB are stored permanently, just like the objects you store in local storage. However, you can delete a database using the `window.indexedDB.deleteDatabase()` method, and some browsers allow users to review how much data each website is storing and remove whatever they don't want.

Storing Records in the Database

Database geeks use the term *data manipulation* to describe every database operation that works with table data, whether it involves reading, changing, inserting, or updating information. In the IndexedDB world, data manipulation operations always follow the same four basic steps. Once you understand this pattern, you'll have an easier time following the code for different database tasks.

Here are the steps:

1. **Create a transaction.**

   Whenever you want to do anything with an IndexedDB database, whether it's writing data or reading it, you must first begin by creating a transaction. In a database, a *transaction* is one or more data operations that are committed together, as a unit. If any part of a transaction fails, every operation inside the transaction is reversed, restoring the database to its pre-transaction state.

Transactions are important in the IndexedDB world because there are so many ways that a database operation can be interrupted. For example, if the user closes the browser while the page is still at work, the browser interrupts your code mid-task. The transaction system guarantees that even when these rude interruptions occur, the database is kept in a consistent state. In fact, transactions are so important that there's no way to perform IndexedDB operations without one.

NOTE To understand the problems you can run into without transactions, imagine you're transferring money from one bank account to another. In a transactional world, the first step (removing money from account A) and the second step (depositing it in account B) either succeed or fail together. But without transactions, you could end up in an unhappy situation: a successful debit from account A, but a failed deposit to account B, leading to some serious missing cash.

2. **Retrieve the object store for your transaction.**

   Object store is a fancy name for table. Because every record in an IndexedDB table is actually a data object, this name makes sense.

3. **Call one of the object store methods.**

   The object store is the gateway to all the table-manipulation features. For example, to add a record you call its `put()` method. To delete a record, you call its `delete()` method. The method returns a request object, which you must use for the next step.

4. **Handle the success and error events.**

   As you already know, virtually everything in IndexedDB is asynchronous. If you want to do something when an operation is finished, and if you want to catch errors before they metastasize into more serious problems, you need to handle the `onSuccess` and `onError` events, just as you did when opening the database.

With these steps in mind, you're ready to look at the code in the `addLink()` function, which runs when the web page visitor clicks the Add Link button.

Before you do anything with your database, you need your data on hand. In the link tracker example, the code grabs the typed-in data from the form and uses it to create a `LinkRecord` object. This task is basic JavaScript—you simply need to find your elements, pull out the corresponding values, and use the `LinkRecord()` function shown on page 342.

```
function addLink() {
  // Collect the data from the form.
  var name = document.getElementById("name").value;
  var url = document.getElementById("url").value;
  var description = document.getElementById("description").value;
  var notes = document.getElementById("notes").value;
```

```
    // Create the LinkRecord object.
    var linkRecord = new LinkRecord(name, url, description, notes);
```

Now you're ready to follow step 1 from the list and create your transaction. To do that, you call the `database.transaction()` method and provide two parameters:

```
var transaction = database.transaction(["Links"], "readwrite");
```

The first parameter is a list of all the tables involved in the transaction. This information enables IndexedDB to lock the tables, preventing other pieces of code from making overlapping and possibly inconsistent changes at the same time. Technically, this parameter is an array that holds a list of table names, which is why you wrap the whole thing in square brackets. But in this example, there's just one table involved in the task.

The second parameter is a string that identifies the type of transaction. Use the word `readwrite` if you want to create a transaction that changes the table in any way, whether it's inserting, updating, or deleting records. But if all you need to do is retrieve data, use the word `readonly` instead.

This brings you to step 2—getting the indispensable object store for your table. You can do so easily with the `transaction.objectStore()` property. Just supply the name of the table, like this:

```
var objectStore = transaction.objectStore("Links");
```

To add a record, use the `put()` method of the object store and supply the data object:

```
var request = objectStore.put(linkRecord);
```

Finally, you need to handle the `onError` and `onSuccess` events to find out whether the record was successfully added:

```
    request.onerror = function(event) {
      alert(request.error + " occurred.");
    };

    request.onsuccess = function(event) {
      // The record has been added.
      // Refresh the display. (For better performance, you could add just the
      // one new item, rather than refresh the whole list.)
      showLinks();
    };
  }
```

NOTE If you call `put()` to add a record that has the same primary key as an existing record (for example, a link with an existing URL), the browser quietly replaces the existing record with your new data.

Querying All the Records in a Table

Querying is the essential job of retrieving your stored information. You pick out a single record, search for a group of records that match certain criteria, or travel through all the records in your table, whichever you prefer.

The link tracker performs a complete table scan and a single-record search. It uses the table scan to create the list of links that appears under the link-adding form. It uses the record search to get the full details for a specific site when you click one of the Details links in the list, as you'll see shortly.

The first task is the more complex one. That's because you need the help of a cursor to browse through an IndexedDB table. (A database *cursor* is an object that keeps track of your current position in a table and lets you step through its records.)

You begin by creating a transaction and getting the object store. This time, you don't need to make any changes, so a read-only transaction fits the bill:

```
function showLinks() {
  var transaction = database.transaction(["Links"], "readonly");
  var objectStore = transaction.objectStore("Links");
```

Next, you create the cursor using the `openCursor()` method of the object store:

```
  var request = objectStore.openCursor();
```

Then, you attach your code to the familiar `onError` and `onSuccess` events. The `onError` event handler is nothing special:

```
  request.onerror = function(event) {
    alert(request.error + " occurred.");
  };
```

The `onSuccess` event handler is more interesting. It has the job of stepping through the records in the Links table, one by one. As it travels, it builds up a string of HTML with the list of links.

```
  // Prepare the string of markup that will be inserted into the page.
  var markupToInsert = "";

  request.onsuccess = function(event) {
    // Create a cursor.
    var cursor = event.target.result;
```

Initially, the cursor is positioned on the first record of the table, if it exists. You check for data by testing whether the cursor is `true` or `false`. If it's `true`, there's a record there ready for you to read. You can get the record from the `cursor.value` property:

```
    if (cursor) {
      var linkRecord = cursor.value;
```

Your data is returned to you as an object. In the link tracker example, each record is a genuine `LinkRecord` object, complete with the `name`, `url`, `description`, and `notes` properties outlined in the object-definition function on page 342.

Once you retrieve your object, it's up to you to decide what to do with it. In the current example, the code uses the `LinkRecord` data to construct an `<a>` element. It uses the site name for the link text and the URL for the link address:

```
markupToInsert += "<a href=" + linkRecord.url + ">" + linkRecord.name +
  "</a>";
```

For now, the `<a>` is stored in a variable called `markupToInsert`. When the code is finished examining every `LinkRecord` in the database and adding all their information to the `markupToInsert` variable, it will finally be ready to copy the markup to the page.

The link tracker example gets a bit fancier by adding two clickable bits of text after every link. One is named "Details" and the other is named "Delete," and you can see them both in Figure 10-7 (near the very bottom of the screen).

These bits of text look like ordinary links, but they are actually `<span>` elements that are hard-wired to call two other JavaScript functions in the page (`getLinkDetails` and `deleteLink`) when clicked. Here's the code that creates them:

```
markupToInsert += "(" +
  "<span class='linkButton' data-url='" + linkRecord.url +
  "' onclick='getLinkDetails(this)'>Details</span>" + " " +
  "<span class='linkButton' data-url='" + linkRecord.url +
  "' onclick='deleteLink(this)'>Delete</span>" +
  ")<br>";
```

There's a neat trick here. The `Details` and `Delete` commands are `<span>` elements. To simplify life, the URL of the corresponding element is stored in each `<span>` element using an attribute. That way, when one of these commands is clicked, the page can quickly retrieve the URL and use it to look up the corresponding record in the Links table.

NOTE The attribute that stores the URL is named `data-url`, according to HTML5 convention. The `data-` prefix indicates that you're using the attribute to hold custom data, which the browser can ignore. You can use whatever you want for the rest of the attribute name—here, `url` makes sense because the attribute is storing a URL.

So far, you've seen how the code processes a single record during a search. When you're ready to move to the next record, you call the `cursor.continue()` method. However, you *don't* attempt to process any more data. That's because stepping through your records is an asynchronous operation. When the cursor reaches the next record, the `onSuccess` event fires again, triggering the same code a second time, at which point you can add the markup for the next record, if it exists.

```
    cursor.continue();
}
```

When you reach the last record in the table, the cursor will evaluate to `false`. At this point, it's time to copy your markup into the page:

```
    else {
      // If there wasn't at least one result, substitute some placeholder text.
      if (markupToInsert == "") {
        markupToInsert = "<< No links. >>";
      }
      else {
        markupToInsert = "<i>Links you've added so far: </i><br>" +
          markupToInsert;
      }

      // Insert the markup.
      var resultsElement = document.getElementById("links");
      resultsElement.innerHTML = markupToInsert;
    }
  };
}
```

Querying a Single Record

Querying an individual record in a table is easier than getting them all, because you don't have to mess around with cursors. Instead, you follow the well-established four-step procedure you saw on page 346, using the `get()` method from the object store.

If you click one of the "Details" links in the link tracker example, the following code springs into action. It grabs the corresponding `LinkRecord` object and extracts all of its information.

```
function getLinkDetails(element) {
  // Get the URL for this link from the handy data-url attribute we added
  // earlier.
  var url = element.getAttribute("data-url");

  var transaction = database.transaction(["Links"], "readonly");
  var objectStore = transaction.objectStore("Links");

  // Find the record that has this URL.
  var request = objectStore.get(url);

  request.onerror = function(event) {
    alert(request.error + " occurred.");
  };

  request.onsuccess = function(event) {
    var linkRecord = request.result;
```

```
        var resultsElement = document.getElementById("linkDetails");
        resultsElement.innerHTML = "<b>" + linkRecord.name + "</b><br>" +
          "<b>URL:</b> " + linkRecord.url + "<br>" +
          "<b>Description:</b> " + linkRecord.description + "<br>" +
          "<b>Notes:</b> " + linkRecord.notes;
      }
    }
```

The information from the `LinkRecord` object is used to create a snippet of HTML markup, which is then inserted into the page in a separate box under the link list. Figure 10-8 shows the result.

Deleting a Record

By this point, you're familiar with the four-step sequence that underpins every data operation. Deleting a record isn't any different. You simply need to use the `delete()` method of the object store.

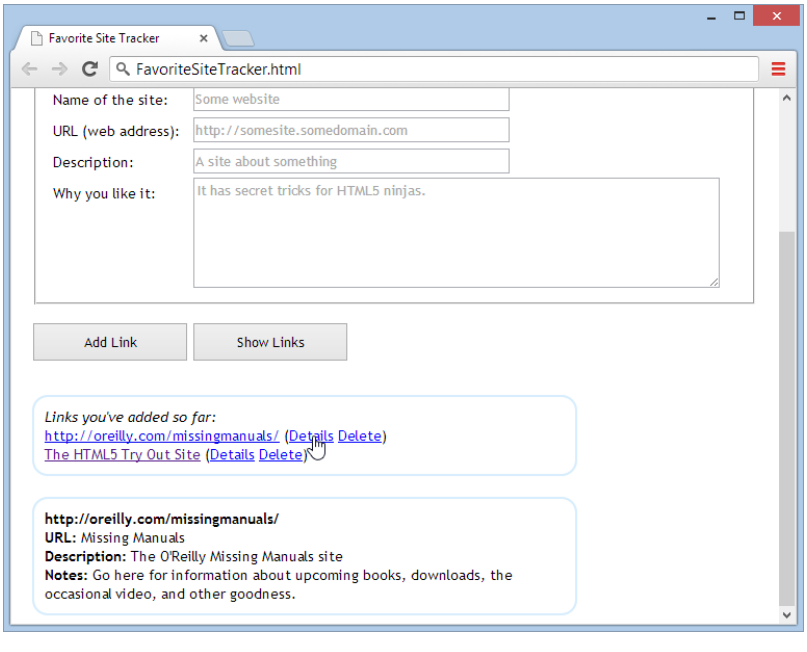

FIGURE 10-8
When you click Details, the code grabs the corresponding record and shows all its data in the bottom box.

In the link tracker example, the "Delete" link does the job. When clicked, it removes the `LinkRecord` with the matching URL, using this code:

```
function deleteLink(element) {
  var url = element.getAttribute("data-url");

  var transaction = database.transaction(["Links"], "readwrite");
  var objectStore = transaction.objectStore("Links");

  var request = objectStore.delete(url);

  request.onerror = function(event) {
    alert(request.error + " occurred.");
  };

  request.onsuccess = function (event) {
    // The record has been deleted.
    // Refresh the display.
    showLinks();
  }
}
```

You've now seen the key methods that you can use to manipulate data with IndexedDB. You've learned how to use the object store to add records or update them (with the `put()` method) and delete them (with `delete`). You've also learned how to retrieve individual records by key value (with `get`) or browse through them all (with `openCursor`). But if you want to get deeper into databases, check out Mozilla's helpful documentation for the object store at *http://tinyurl.com/objectstore*, or explore the nitty-gritty details of the IndexedDB standard at *www.w3.org/TR/IndexedDB*.

Browser Support for IndexedDB

IndexedDB is a relatively new specification,and requires a relatively recent browser. Table 10-2 shows the current state of affairs.

TABLE 10-2 *Browser support for IndexedDB*

	IE	FIREFOX	CHROME	SAFARI	OPERA	SAFARI IOS	CHROME FOR ANDROID
Minimum version	10	10	23	-	15	-	29

Sadly, IndexedDB isn't yet available on desktop or mobile versions of Safari. This support gap exists because the developers of Safari invested their effort in the now-abandoned Web SQL Database standard. If you need Safari support, you can use a polyfill that converts IndexedDB operations to the corresponding Web SQL commands. (Download it at *http://tinyurl.com/DBpolyfill*.) But if you want IndexedDB support on a browser that doesn't support IndexedDB or Web SQL, such as Internet Explorer version 9 or 8, you're unfortunately out of luck.