A Brain-Friendly Guide

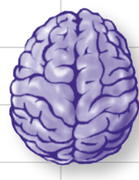
Head First HTML5 Programming

A learner's guide to building web apps with JavaScript

Learn the secrets of the HTML5 guru

Discover why everything your friends know about video is probably wrong

Avoid embarrassing browser support issues



Load HTML5 and JavaScript straight into your brain

Watch out for common browser pitfalls

O'REILLY®

Eric Freeman & Elisabeth Robson

Head First HTML5 Programming

A JavaScript CloseUp

Let's take a closer look at `fillStyle` since this is the first time you've seen it. `fillStyle` is a property of the context that holds the color for any drawing you do to the canvas.

Just like fillRect, fillStyle is something we control through the context.

But unlike fillRect, fillStyle is a property, not a method. So we set it, rather than call it.

And what we set it to is a color. You can use the same color formats you use in CSS. So you can use color names, like lightblue, or values like #ccccff or rgb(0, 173, 239). Try it!

```
context.fillStyle = "lightblue";
```

Note that unlike in CSS, you must put quotes around the value if you're not using a variable.

there are no Dumb Questions

Q: I was expecting we'd set the background color of the squares and the canvas by passing in a color value to fillRect. I don't really get how fillStyle works. How does it affect what fillRect does?

A: Great question. This is a little different from how you might be used to thinking of things. Remember, the context is an object that controls access to the canvas. What you're doing with fillStyle and fillRect is first setting a property that tells the canvas, "Whatever you draw next should be in this color". So anything you fill with color (like with fillRect) after setting the fillStyle will use that color, until you change the color again by setting fillStyle to a different color.

Q: Why does the color need quotes around it, when the property values in CSS don't? I don't use quotes when I'm setting the background-color of an element, for instance.

A: Well, CSS is a different language from JavaScript, and CSS doesn't expect quotes. But if you don't use quotes around the color, JavaScript will think that the color name is a variable instead of a string, and will try to use the value of the variable for the color.

Say you have a variable `fgColor = "black"`. You could write `context.fillstyle = fgColor`, and it would work because the value of fgColor is "black".

But `context.fillStyle = black` won't work because black isn't a variable (unless you set it up that way, which might be a bit confusing). You'll know you've made this mistake because you'll get a JavaScript error that says something like "Can't find variable: black". (Don't worry, we all make that mistake at least once.)

Q: Okay, I give up. Why were we seeing less than 20 squares sometimes?

A: The x, y and width of the squares are all random. Some squares might obscure other squares. A square might have an x, y position of 599, 199 so you'd only be able to see one pixel of that square (because the rest of the square would be off the canvas). Some squares might be 1 pixel wide, and some squares might even be 0 pixels wide because the Math.random method can return 0. Or you might generate two squares of exactly the same size and location.

But for this application it's all part of the randomness, so we think it's fine. For another application we might need to ensure this doesn't happen.

Meanwhile, back at TweetShirt.com...

Jim: I know, and I'm impressed with how little code this took. Just think if we did this the old, server-side way, we'd still be getting our server up.

Frank: And it seems like we're in a good position to knock out the circles in the design too; after all, they are just like the squares.

Jim: I agree, where's Judy? She probably knows the API for the circles already. Then again, it's probably just a matter of calling the `fillCircle` method.

Frank: Sounds right to me! Who needs Judy, we've got this!

And, a couple of hours later...

Frank: I don't know what's going on, I've double checked everything, but, no matter what I do, when I call `fillCircle`, I get nothing on the canvas.

Judy: Well, show me your `fillCircle` method.

Frank: What do you mean by my method? I don't have one, I'm using the method in the canvas's API directly.

Judy: The canvas API doesn't have a `fillCircle` method.

Frank: Er, I *assumed* it did because we have a `fillRect`...

Judy: Yeah, well you know what that ass-u-me stuff gets us. Come on, pull up a browser—you can always find the API at:
`http://dev.w3.org/html5/2dcontext/`.

...Anyway, drawing a circle is a little more complex than calling a single method. You need to learn about paths and arcs first.

Jim, entering: Judy, did Frank tell you about how we nailed the circle?

Frank: Uh yeah, Jim, *enoughway ithway ethay irclecay!*

We recommend the translation services of piglatin.bavetta.com.

Drawing with Geeks

Before digging into circles, we need to talk about paths and arcs. Let's start with paths, and draw some triangles. If we want to draw a triangle on the canvas, there's no `fillTriangle` method, but we can create a triangle anyway by first creating a *path* in the shape of a triangle, and then *stroking* over it to draw a triangle on the canvas.

What does that mean? Well, say you wanted to be really careful drawing on the canvas, you might take a pencil and trace a light shape (let's just call it a path) on the canvas. You trace it so lightly that only you can see it. Then, after you're satisfied with your path, you take a pen (with a thickness and color of your choosing) and you use that to stroke over the path so everyone can see your triangle (or whatever shape you traced with the pencil).

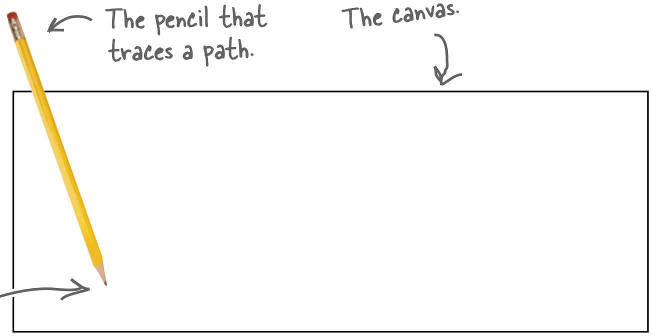
That's just how drawing arbitary shapes with lines works on the canvas. Let's draw a triangle and see how this works:

We use the beginPath method to tell the canvas we're starting a new path.

```
context.beginPath();
context.moveTo(100, 150);
```

We use the moveTo method to move the "pencil" to a specific point on the canvas. You can think of the pencil as being put down at this point.

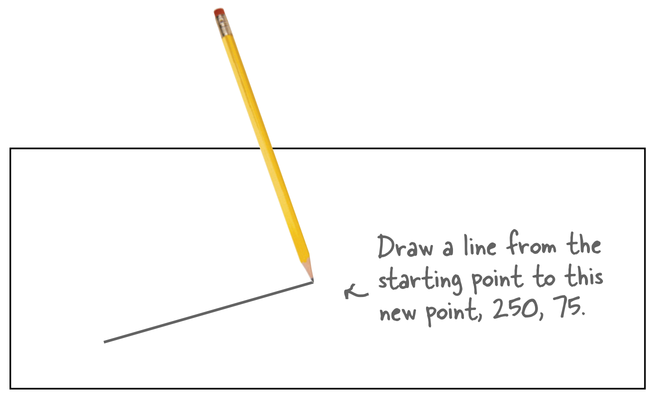
Here we're putting the pencil down at x = 100 and y = 150. This is the first point on the path.



The lineTo method traces a path from the pencil's current location to another point on the canvas.

```
context.lineTo(250, 75);
```

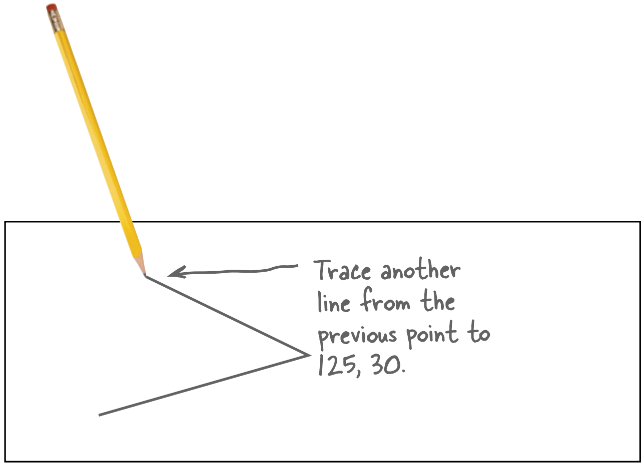
The pencil was at 100, 150, and here we're extending the path from there to the point x=250, y=75.



We've got the first side of the triangle, now we need two more. Let's use lineTo again for the second side:

```
context.lineTo(125, 30);
```

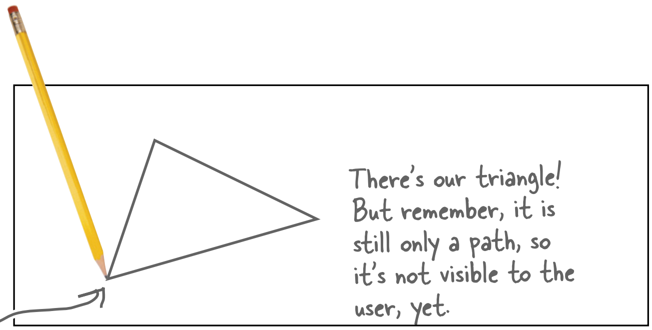
Here we're tracing from the current pencil position (250, 75) to a new position, x = 125, y = 30. That completes our second line.



We're almost there! All we need to do now is to trace one more line to finish the triangle. And to do that, we're just going to close the path with the closePath method.

```
context.closePath();
```

The closePath method connects the starting point of the path (100, 150) to the last point in the current path (125, 30).



Exercise

So you have a path! Now what?

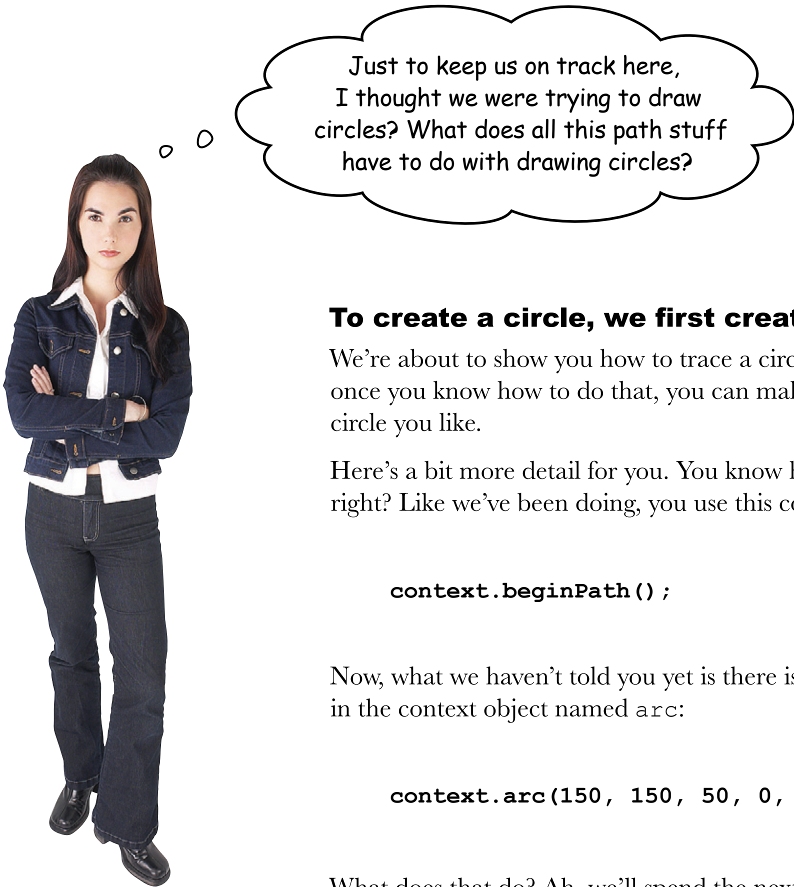
You use the path to draw lines and fill in your shape with color, of course! Go ahead and create a simple HTML5 page with a canvas element and type in all the code so far. Give it a test run, too.

```
context.beginPath();
context.moveTo(100, 150);
context.lineTo(250, 75);
context.lineTo(125, 30);
context.closePath();
```

Here's the code so far.

```
context.lineWidth = 5;          ..............................

context.stroke();               ..............................

context.fillStyle = "red";      ..............................

context.fill();                 ..............................
```

And here's some new code. Go ahead and annotate this with what you think it does. Load the page. Were you right? Check your answers at the end of the chapter.



To create a circle, we first create a path.

We're about to show you how to trace a circle as a path. And, once you know how to do that, you can make any kind of circle you like.

Here's a bit more detail for you. You know how to start a path, right? Like we've been doing, you use this code:

```
context.beginPath();
```

Now, what we haven't told you yet is there is another method in the context object named `arc`:

```
context.arc(150, 150, 50, 0, 2 * Math.PI, true);
```

Do you happen to remember from geometry class that the circumference of a circle = 2πR? Just put that in the back of your mind for a sec...

What does that do? Ah, we'll spend the next page or so finding out the details. But, as you might expect, it traces a path along a circle.

Breaking down the arc method

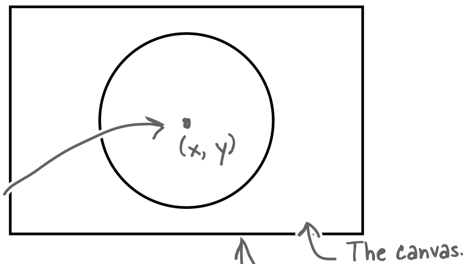
Let's dive right into the arc method and check out its parameters.

```
context.arc(x, y, radius, startAngle, endAngle, direction)
```

The whole point of the arc method is to specify how we want to trace a path along a circle. Let's see how each parameter contributes to that:

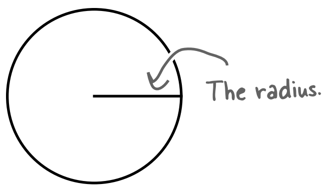
`x,y` The x and y parameters determine where the center of the circle will be located in your canvas.

This is the x, y position of the center of your circle.

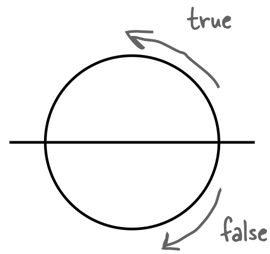



```
context.arc(x, y, radius,
```

`radius` This parameter is used to specify 1/2 the width of the circle.



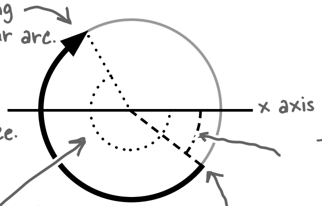
`direction` determines if we are creating the arc path in a counterclockwise or clockwise direction. If direction is true, we go counterclockwise; if it's false, we go clockwise.



If direction is true, we trace the arc going counter-clockwise. If it is false we trace clockwise.

```
startAngle, endAngle, direction)
```

`startAngle, endAngle` The start angle and end angle of the arc determine where your arc path starts and stops on the circle.



The stopping point of our arc.

The arc we want to trace.

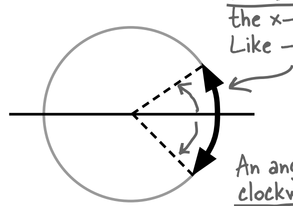
The end angle is the angle between the x-axis and the stopping point of our arc.

The starting point of our arc.

The start angle is the angle between the x axis and the starting point of the arc.

Important Point Below!

Don't skip this. Angles can be measured in the negative direction (counterclockwise from the x-axis) or in the positive direction (clockwise from the x-axis). This is *not* the same as the direction parameter for the arc path! (You'll see on the next page.)

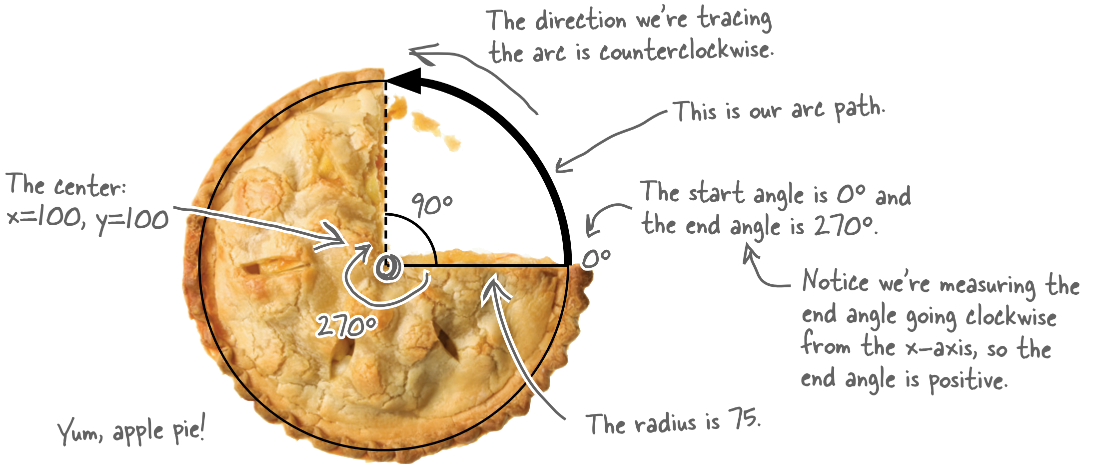


An angle measured going counterclockwise from the x-axis is negative. Like -35 degrees.

An angle measured going clockwise from the x-axis is positive. Like 45 degrees.

A little taste of using the arc

What we need right now is a good example. Let's say that you want to trace an arc over a circle that is centered at x=100, and y=100 and you want the circle to be 150 pixels wide (or, a radius of 75). And, the path you want to trace is just 1/4 of the circle, like this:



Now let's create an arc method call that traces this path:

1. We start with the x, y point of the center of the circle: 100, 100.

```
context.arc(100, 100, __, __, ______________________, ____);
```

2. Next, we need the radius of the circle, 75.

```
context.arc(100, 100, 75, __, ______________________, ____);
```

3. What about our start and end angles? Well, the start angle is 0 because the starting point is at 0 degrees from the x axis. The end angle is the angle between the x-axis and the stopping point of our arc. Since our arc is a 90 degree arc, our end angle is 270 degrees (90+270 = 360). (Note that if we'd measured in the negative, or counterclockwise direction instead, our end angle would be -90 degrees.)

```
context.arc(100, 100, 75, 0, degreesToRadians(270), ____);
```

We'll come back to this in a sec. It just converts degrees (which we're used to), into radians (which the context seems to prefer).

4. Finally, we're tracing the arc in a counterclockwise direction, so we use `true`.

```
context.arc(100, 100, 75, 0, degreesToRadians(270), true);
```

I say degree, you say radian

We all talk about circle angles every day: "nice 360 dude," or "I was heading down that path and I did a complete 180," or, ...well, you get the picture. The only problem is, we think in *degrees*, but the canvas context thinks in *radians*.

A radian is just another measure of an angle. One radian equals 180/3.14159265... (or 180 divided by π).

Now, we could tell you that:

```
360 degrees = 2Pi radians
```

and you're good to go if you want to compute radians in your head from now on. Or, if for some reason you'd rather not do that in your head, here's a handy function that will do the work for you:

```
function degreesToRadians(degrees) {
    return (degrees * Math.PI)/180;
}
```

Use this function whenever you want to think in degrees, but get radians for drawing an arc.

You might remember seeing this briefly in the Geolocation chapter.

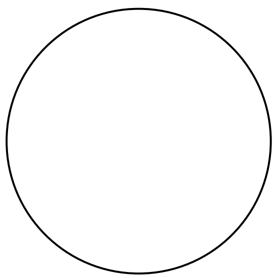
To get radians from degrees, you multiply by π and divide by 180.

On page 313, you saw us use 2*Math.PI to specify the end angle of a circle. You could do that... or just use degreesToRadians(360).

BE the Browser

Interpret the call to the arc method and sketch out all the values on the circle, including the path the method creates.

```
context.arc(100, 100, 75, degreesToRadians(270), 0, true);
```



Annotate this circle with all the arguments and then draw the path this method call creates.

Hint: what is left in the pie after eating this?

Back to writing the TweetShirt circle code

Now that you know how to draw circles, it's time to get back to TweetShirt and add a new function, `drawCircle`. We want to draw 20 random circles, just like we did for squares. To draw those circles we need to first determine that the user has selected circles from the shape menu. Let's add that to the previewHandler function.

Edit your `tweetshirt.js` file and add the new code below.

```
function previewHandler() {
    var canvas = document.getElementById("tshirtCanvas");
    var context = canvas.getContext("2d");
    fillBackgroundColor(canvas, context);

    var selectObj = document.getElementById("shape");
    var index = selectObj.selectedIndex;
    var shape = selectObj[index].value;

    if (shape == "squares") {
        for (var squares = 0; squares < 20; squares++) {
            drawSquare(canvas, context);
        }
    } else if (shape == "circles") {
        for (var circles = 0; circles < 20; circles++) {
            drawCircle(canvas, context);
        }
    }
}
```

This code looks almost identical to the code to test for squares. If the user has chosen circles rather than squares then we draw 20 circles with the drawCircle function (which we now need to write).

We're passing the canvas and context to the drawCircle function, just like we did with drawSquares.

BRAIN POWER

What start angle and end angle do you use to draw a complete circle?

What direction do you use: counterclockwise or clockwise? Does it matter?

A: You draw a circle with a start angle of 0° and an end angle of 360°. It doesn't matter what direction you use since you're drawing a complete circle.

Writing the drawCircle function...

Now let's write the `drawCircle` function. Remember, here we just need to draw one random circle. The other code is already handling calling this function 20 times.

```
function drawCircle(canvas, context) {
    var radius = Math.floor(Math.random() * 40);
    var x = Math.floor(Math.random() * canvas.width);
    var y = Math.floor(Math.random() * canvas.height);


    context.beginPath();
    context.arc(x, y, radius, 0, degreesToRadians(360), true);

    context.fillStyle = "lightblue";
    context.fill();
}
```

Just like we did for squares, we're using 40 for the maximum radius size to keep our circles from getting too big.

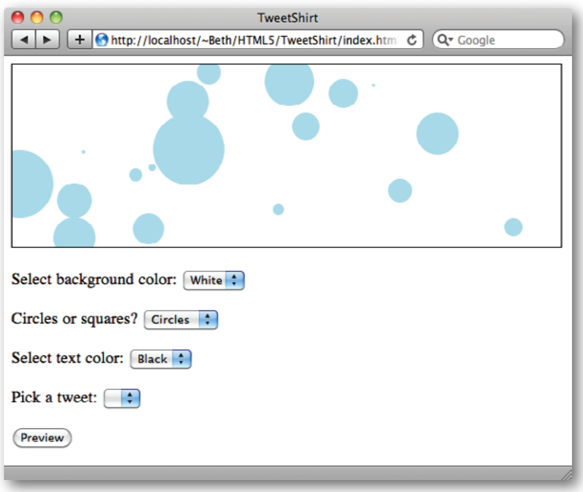
And, again, the x & y coordinates of the center of the circle are based on the width and height of the canvas. We choose random numbers between 0 and the width and height respectively.

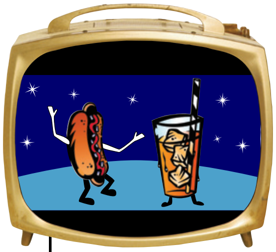
We use an end angle of 360° to get a full circle. We draw counterclockwise around the circle, but for a circle, it doesn't matter which direction we use.

We're using "lightblue" as our fillStyle again, and then filling the path with context.fill().

... and test drive!

So go ahead and type this in (and don't forget to add the degreesToRadians function too), save, and load it in your browser. Here's what we see (given these are random circles—yours will look a little different):





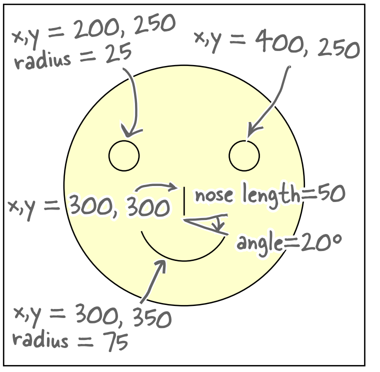
Intermission

A Little Cookie Break

Whew! That was a fun set of pages we just went through. We don't know about you, but we're ready for some cookies. How about taking a short cookie break? But, don't think we aren't going to give you something fun to do while you're having them (check on the exercise to the right).

So, sit back, have a little break, and nibble on this while you give your brain and stomach something else to do for a bit. Then come on back and we'll finish off the TweetShirt code!

To the right you'll find a smiley face (or a chocolate chip cookie smiley face if you prefer). The code below to draw the smiley face is almost done; we just need your help to finish it up. With all the work you've done in this chapter, you've got all you need to complete this. You can always check your answer at the end of the chapter when you're finished.



Here's what we're looking for. You might want to make some real chocolate chip cookie smiley faces while you're at it...

```
function drawSmileyFace() {
    var canvas = document.getElementById("smiley");
    var context = canvas.getContext("2d");

    context.beginPath();
    context.arc(300, 300, 200, 0, degreesToRadians(360), true);
    context.fillStyle = "#ffffcc";
    context.fill();
    context.stroke();

    context.beginPath();
    context.arc(____, ____, 25, ____, _________, true);
    context.stroke();

    context.beginPath();
    context.arc(400, ____, ____, ____, ________, _____);
    context.stroke();

    context.beginPath();
    context._______(____, ____);
    context._______(____, ____);
    context.________();

    context.beginPath();
    context.____(300, 350, ____, degreesToRadians(____), degreesToRadians(____), _____);
    context.stroke();
}
```

The face circle. We did this one for you. Notice we filled it with yellow.

This is the left eye.

This is the right eye.

This is the nose.

And here's the mouth. This is the tricky one!

Welcome back...

You're back, you're rested, refreshed, and we're on the home stretch to getting this start-up off the ground. When you look at all the work we've done, all we really have left is to display the tweets and the other text in the canvas preview.

Now, to get a get a tweet on the canvas we first need some of your recent tweets to choose from, and we're going to use JSONP to do it. If you remember Chapter 6, you already know how to do this. If you need to, go back to Chapter 6 for a quick refresher. All we're going to do is:

1. Add a <script> at the bottom of the tweetshirt. html file to make a call to the Twitter JSONP API. We're going to ask for the most recent status updates of a specified user.

2. Implement a callback to get the tweets that Twitter sends back. We'll use the name of this callback in the URL for the <script> in Step 1.

Here's our HTML file for TweetShirt.

Imagine your head element here, and your form here (we wanted to save a few trees).

Here's our JSONP call; this works by retrieving the JSON created by calling the Twitter URL, and then passing that JSON to the callback function (which we'll define in just a sec).

Here's the Twitter API call. We're asking for a user timeline which will give us recent statuses.

Change this to your username, or someone else's if you like.

```
<html>
...
<body>
    <form>
    ...
    </form>
    <script src="http://twitter.com/statuses/user_timeline/wickedsmartly.json?
callback=updateTweets">
    </script>
</body>
</html>
```

And here's the callback function where the JSON will be passed back.

Type this all on one line in your text file (it's too long to fit on one line in the book).

There's a lot going on here. If this is only vaguely familiar, please do have a look back at how JSONP works, in Chapter 6.

Getting your tweets

We've already done the hard work, which is getting the tweets from Twitter. Now we need to add them to the tweets `<select>` element in the `<form>` of our page. Just to review again: when the callback function is called (in our case, the function `updateTweets`), Twitter hands it a response that contains JSON formatted tweets.

Twitter's response is an array of tweets. Each tweet has a ton of data in it; the piece we're going to use is the text of the tweet.

Edit your `tweetshirt.js` file and add the `updateTweets` function at the bottom. Here's the code:

Here's our callback.

Which is passed a response containing the tweets from the user timeline as an array of tweets.

We grab a reference to the tweets selection from the form.

```
function updateTweets(tweets) {
    var tweetsSelection = document.getElementById("tweets");

    for (var i = 0; i < tweets.length; i++) {
        tweet = tweets[i];
        var option = document.createElement("option");
        option.text = tweet.text;
        option.value = tweet.text.replace("\"", "'");

        tweetsSelection.options.add(option);
    }

    tweetsSelection.selectedIndex = 0;
}
```

For each tweet in the array of tweets, we:

Get a tweet from the array.

Create a new option element.

Set its text to the tweet.

And set its value to the same text, only we've processed the string a little to replace double quotes with single quotes (so we can avoid formatting issues in the HTML).

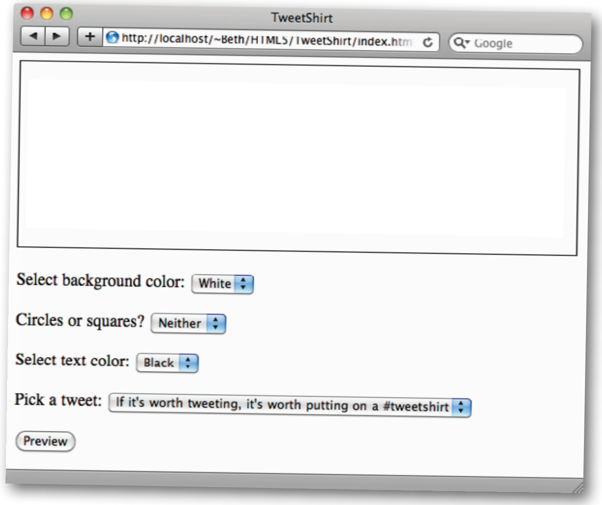
We then take the new option, and add it to the tweet selection in the form.

And, finally, we make sure the first tweet is the selected tweet by setting the selectedIndex of the <select> to 0 (the first option element contained within it).

After we've done this for each tweet, we have a <select> element that contains an option for each tweet.

Test driving Tweets

Let's do a quick test drive. Make sure you've got all the code added to `tweetshirt.js` and `index.html`. Also make sure you're using a Twitter username that has recent tweets in your script src URL (so you'll be sure you see some tweets!). Load the page and click on the tweets selection. Here's what we see:



Here's the tweets menu with REAL tweets in it. Cool!

Guys, this is great. We have squares and circles and Jim's got us hooked up to Twitter! What's next?

Frank's tablet.

Jim: We're almost there. We need to nail down all the text we need to display. We've got the two messages: the "I saw this tweet" and "and all I got was this lousy t-shirt!" and also the tweet the user has chosen to display. Now we've got to figure out how to display it, not to mention apply some styling to the text.

Frank: I'm assuming we can throw some text in the canvas and then apply some CSS to it?

Joe: I don't think it works like that. The canvas is a drawing area, I don't think we can place text and style it, we have to draw text onto the canvas.

Jim: Well, this time I learned my lesson and I've already checked out the API for text.

Joe: Good, I haven't looked yet; how does it look?

Jim: Remember that arc method? We have to custom draw all our text using that.

Frank: Are you kidding me? I guess I'll be working all weekend now.

Jim: Gotcha! No seriously, there is a `fillText` method that takes the text to draw on the canvas along with the x, y position of where to draw it.

Joe: That sounds pretty straightforward. What about the differences in style? If I remember the comp, the tweet text is italic, and the rest of the text is bold.

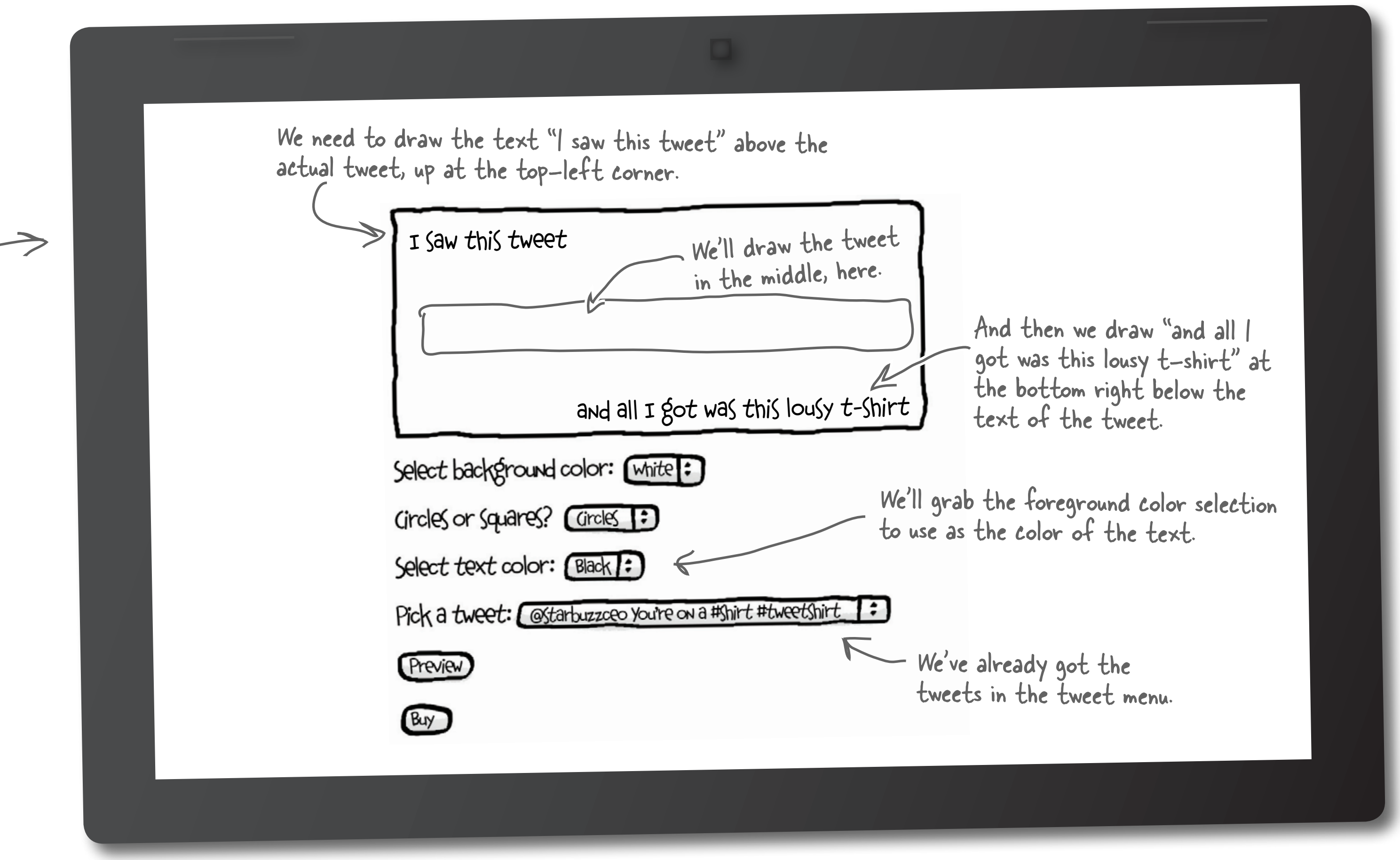
Jim: We need to look a bit more, there are various methods for setting alignment and fonts and fill styles, but I'm not not quite sure how to use them.

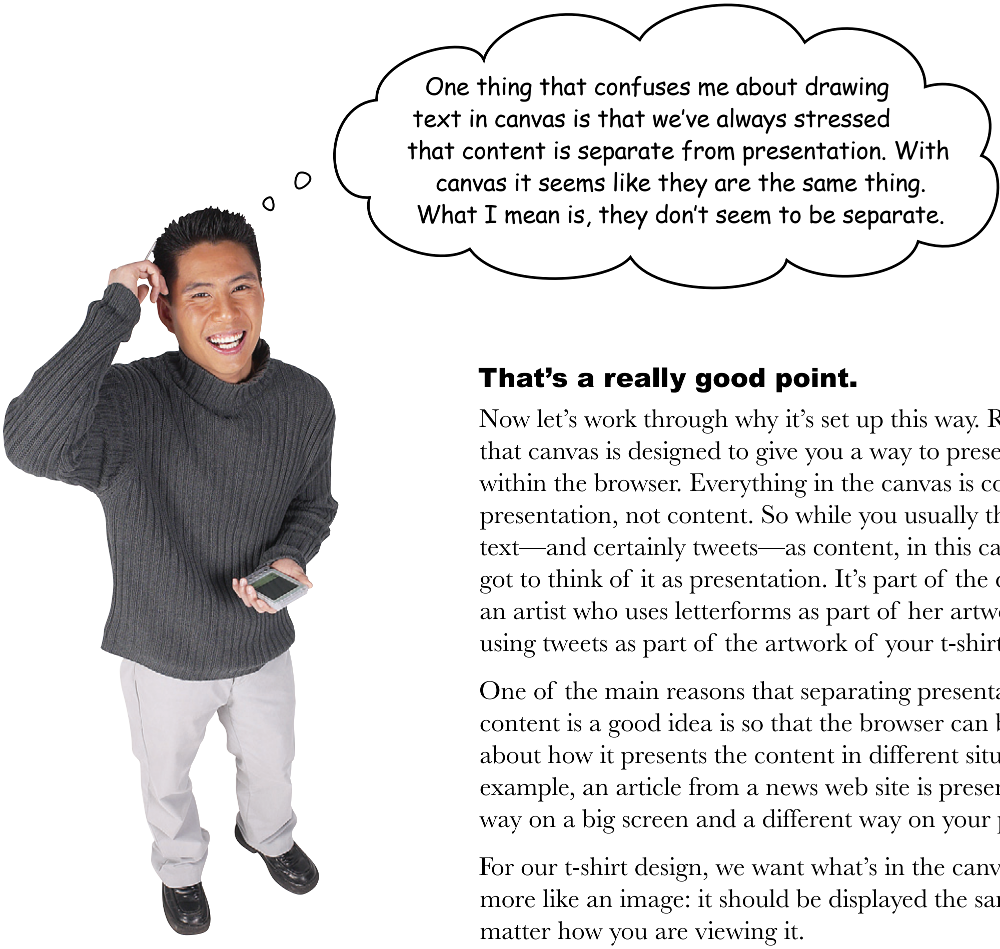
Frank: And to think maybe I could have helped, but no CSS?

Jim: Sorry, like Joe said, this is an API for drawing onto a canvas, it doesn't make use of HTML or CSS styling in any way.

Joe: Well, lets pull up the API and take a look and then we can try getting the "I saw a tweet" text on the canvas. Come on Frank, join us, this can't be too bad and I'm sure we can use your knowledge of fonts, styles and all that.

Frank: Sure thing, I'm here for you!





That's a really good point.

Now let's work through why it's set up this way. Remember that canvas is designed to give you a way to present graphics within the browser. Everything in the canvas is considered presentation, not content. So while you usually think of text—and certainly tweets—as content, in this case, you've got to think of it as presentation. It's part of the design. Like an artist who uses letterforms as part of her artwork, you're using tweets as part of the artwork of your t-shirt design.

One of the main reasons that separating presentation and content is a good idea is so that the browser can be smart about how it presents the content in different situations: for example, an article from a news web site is presented one way on a big screen and a different way on your phone.

For our t-shirt design, we want what's in the canvas to be more like an image: it should be displayed the same way no matter how you are viewing it.

So, let's get the text on the canvas and get this start-up rolling!

Code Magnets

It's time for your first experiment with canvas text. Below we've started the code for drawText, the method we're going to call to draw all the text onto the preview canvas. See if you can finish up the code to draw "I saw this tweet" and "and all I got was this lousy t-shirt!" on the canvas, we'll save drawing the actual tweet for later. Make sure you check your answer with the solution at the end of the chapter before you go on.

```
function drawText(canvas, context) {

    var selectObj = document.getElementById("__________________");

    var index = selectObj.selectedIndex;

    var fgColor = selectObj[index].value;

    context.____________ = fgColor;

    context.__________ = "bold 1em sans-serif";

    context.______________ = "left";

    context.__________(_"___________________, 20, 40);


    // Get the selected tweet from the tweets menu
    // Draw the tweet


    context.font = "______________________";

    context.____________ = "__________";

    context.____________("and all I got was this lousy t-shirt!",

             _____________________, _____________________);

}
```

Hint: This is the x, y position for the "I saw this tweet" text.

For right now, we're just putting comments in where the code to draw the tweet text will be.

Hint: We'll be using an italic serif font for the tweet, but we want this one to be bold sans-serif.

Hint: we want to position the text in the bottom-right corner.

We want to draw this text at 20 from the right side of the canvas, and 40 from the bottom of the canvas, so it balances the top line of text.

Magnets:

- `foregroundColor`
- `fillStyle`
- `fillText`
- `bold 1em sans-serif`
- `fillText`
- `font`
- `textAlign`
- `canvas.width-20`
- `right`
- `textAlign`
- `canvas.height-40`
- `fillCircle`
- `fillRect`
- `"I saw this tweet"`
- `left`

Canvas Text Up Close

Now that you've had a chance to draw your first text in the canvas, it's time to take a closer look at the text methods and properties available in the canvas API. As you found out in the exercise this is a fairly low-level API—you have to tell the context what text to draw, what position to use, and what font to use.

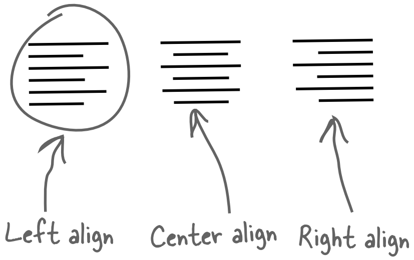
In this Up Close segment we'll examine the alignment, font, and baseline properties, and the fill and stroke methods in detail so you'll be a canvas text expert by the time you turn the page!

alignment

The `textAlign` property specifies where the anchor point for the text is. "start" is the default.

```
context.textAlign = "left";
```

Possible values are: start, end, left, right, and center. Start and end mean the same as left and right in left-to-right languages, like English, and are reversed in right-to-left languages, like Hebrew.



fill and stroke

Just like for rectangles, we can stroke and fill text. We provide the text to draw, the x, y position and an optional parameter, maxwidth, that causes the text to scale if it is wider than maxwidth.

```
context.fillText("Dog", 100, 100, 200);

context.strokeText("Cat", 100, 150, 200);
```

filled text.

Dog

Cat

stroked text.

If the text gets wider than 200, it is automatically scaled to fit.

Font

To set the font properties, you can use the same format you're used to using in CSS, which is handy. If you specify every property value, you'll include: font style, weight, size, and family, in that order.

```
context.font = "2em Lucida Grande";
context.fillText("Tea", 100, 100);
context.font = "italic bold 1.5em Times, serif";
context.fillText("Coffee", 100, 150);
```

Tea

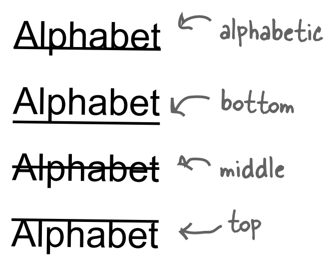
Coffee

The spec recommends that you use vector fonts only (bitmap fonts may not display very well).

baseline

The `textBaseline` property sets the alignment points in the font and determines the line your letters sit on. To see how the line affects the text, try drawing a line at the same x, y point you draw a text.

```
context.beginPath();
context.moveTo(100, 100);
context.lineTo(250, 100);
context.stroke();
context.textBaseline = "middle";
context.fillText("Alphabet", 100, 100);
```



Possible values are: top, hanging, middle, alphabetic, ideographic, and bottom. The default is alphabetic. Experiment with the different values to find what you need (and check out the spec for more details).

Giving drawText a spin

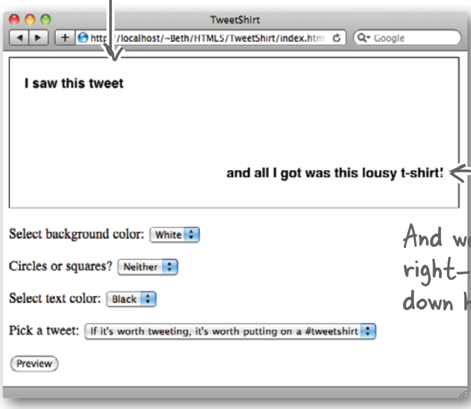
Now that you've got more of the API in your head, go ahead and get the code you created in the Magnet Code exercise typed in— here it is with the magnets nicely translated to code:

```
function drawText(canvas, context) {
    var selectObj = document.getElementById("foregroundColor");
    var index = selectObj.selectedIndex;
    var fgColor = selectObj[index].value;

    context.fillStyle = fgColor;
    context.font = "bold 1em sans-serif";
    context.textAlign = "left";
    context.fillText("I saw this tweet", 20, 40);





    context.font = "bold 1em sans-serif";
    context.textAlign = "right";
    context.fillText("and all I got was this lousy t-shirt!",
        canvas.width-20, canvas.height-40);
}
```

We're going to put the code that draws the tweet text here in a sec...

After you've got it typed in, update your `previewHandler` function to call the `drawText` function, and give it a test drive by loading it in your browser. You should see something like we do:

Here's the text. We've got sans-serif text in bold at the correct location.



And we've got right-aligned text down here.

Sharpen your pencil

Take a shot at completing the drawText function. You need to get the selected tweet, set the font to an italic serif font that's slightly (1.2em) bigger than than the default, make sure the text is aligned left, and position it at x = 30, y = 100. This is the last step before we see the TweetShirt!

Write your code above, and don't peek at the next page! (Really!)

Completing the drawText function

Here's our solution code. How does it compare to yours? If you haven't already typed your code in, go ahead and type in the code below (or your version if you prefer), and reload your `index.html`. We'll show you our test drive on the next page.

```
function drawText(canvas, context) {
    var selectObj = document.getElementById("foregroundColor");
    var index = selectObj.selectedIndex;
    var fgColor = selectObj[index].value;

    context.fillStyle = fgColor;
    context.font = "bold 1em sans-serif";
    context.textAlign = "left";
    context.fillText("I saw this tweet", 20, 40);

    selectObj = document.getElementById("tweets");
    index = selectObj.selectedIndex;
    var tweet = selectObj[index].value;
    context.font = "italic 1.2em serif";
    context.fillText(tweet, 30, 100);

    context.font = "bold 1em sans-serif";
    context.textAlign = "right";
    context.fillText("and all I got was this lousy t-shirt!",
        canvas.width-20, canvas.height-40);
}
```

We don't need to align the tweet text to the left; the alignment is still set from up here.

We grab the selected option from the tweet menu.

Set the font to italic serif, just a tad bigger...

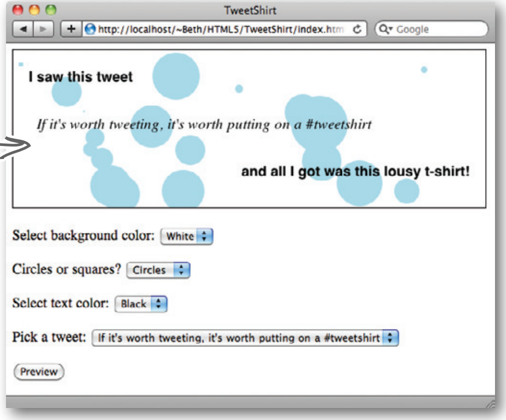
... and draw it at position 30, 100.

A quick test drive and then ~~LUNCH~~ LAUNCH!

We hope you're seeing what we're seeing! Nice huh? Give the interface a real bit of quality assurance testing: try all the combinations of colors and shapes, or swap out the username for another you like.

Feel like you're ready to launch this for real? Let's do it!

There's the tweet on the t-shirt preview. Nice!



Remember the TweetShirt founder?

1. The first thing we need is an image. We've put an image named `twitterBird.png` in the TweetShirt folder. To get that into the canvas we first need a JavaScript image object. Here's how we get one:

```
var twitterBird = new Image();
twitterBird.src = "twitterBird.png";
```

Create a new image object.

And set its source to be the image of the Twitter bird.

2. The next part should feel pretty natural by now; we need to draw the image on the canvas using a context method named, you guessed it, `drawImage`.

```
context.drawImage(twitterBird, 20, 120, 70, 70);
```

Using the drawImage method

Here's our image object.

And we specify the x, y location, width and height.

3. There's one other thing to know about images: they don't always load immediately, so you need to make sure that the image has fully loaded beford you draw it. How do we wait until something is loaded before we take action? We implement an onload handler:

Here, we're saying: when the image has loaded, then execute this function.

```
twitterBird.onload = function() {
    context.drawImage(twitterBird, 20, 120, 70, 70);
};
```

We draw the image to the canvas using the context's drawImage method.

Exercise

See if you can put together the drawBird function from all the pieces Judy gave us. The drawBird function takes a canvas and context, and draws the bird onto the canvas. You can assume that with this function we're placing "twitterBird.png" at the location x=20, y=120, with a width and height of 70. We've written the method declaration and the first line for you. You'll find our solution at the end of the chapter.

```
function drawBird(canvas, context) {
    var twitterBird = new Image();

}
```

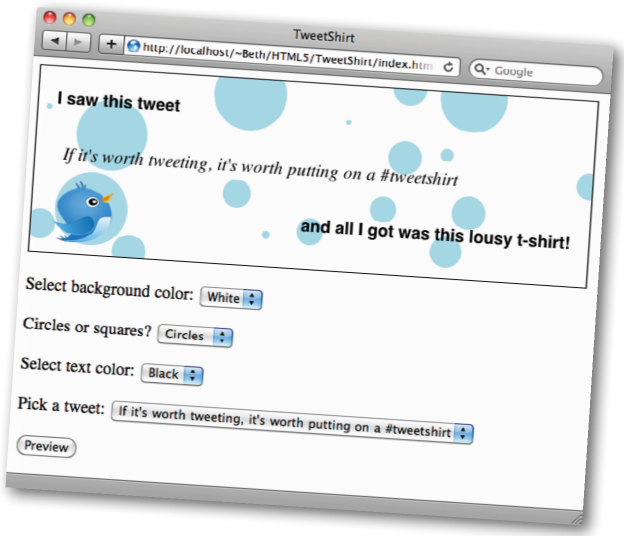
Your code here. →

Make sure you add a call to the drawBird function in the previewHandler function.

Yet another test drive

Double check your code and get in another test drive! Wow, this is really looking polished now.

Give it a few tries; try it with circles or squares. You'll notice that we used a png with a transparent background so that the circles and squares work if they're behind the bird.



This rocks and we're well on our way to developing a cool app. But like we said, we're counting on you to implement the ecommerce and fulfillment and all that.

there are no Dumb Questions

Q: We haven't seen the Image object before. You used it when you added an image to the canvas. What is this? Why didn't we create it with document.createElement("img")?

A: Good catch. Both methods you mention create image objects. The JavaScript Image constructor is arguably a more direct way to create images from JavaScript and gives us a little more control over the loading process (like giving us the ability to easily use a handler to be notified when the image is loaded).

So, our goal here is to create an image, and to make sure it's loaded before we draw it on the canvas. The Image object gives us the best path to doing that.

Q: Canvas is cool... but also kind of a pain compared to HTML. Anything more complicated than basic shapes must be really difficult to do.

A: There's no doubt about it, you're writing graphics code when you're programming canvas. Unlike the browser, which takes care of a lot of details for you, like flowing elements onto the page so you don't have to worry about drawing everything yourself, you have to tell canvas where to put everything.

Canvas gives you a lot of power to do almost any kind of graphics (currently, 2D) you can imagine, however. And we're in early days of canvas; it's likely that libraries of JavaScript code will make it easier to write graphics for canvas in the future.

Q: I noticed that for very long tweets, the tweet just disappears off the edge of the canvas. How can I fix that?

A: One way to fix it is to check to see how many characters the tweet contains and if it's greater than a certain number, split it into multiple lines and draw each line separately onto the canvas. We've included a function, splitIntoLines, in the code on wickedlysmart.com that you can use to do just that.

Q: I also noticed that some tweets have HTML entities in them, like " and &. What's that all about?

A: The Twitter API that we're using to get tweets as JSON converts characters that people post in their tweets to HTML entities. It's actually a good thing because any special characters, or even quotes, that would mess up our ability to get the tweets properly from the JSON are represented as entities. If we were displaying the tweets in HTML, those entities would be displayed in the browser as the characters you're meant to see, just like entities you add to your own page are displayed correctly in the browser. However, as you saw, in the canvas they don't look so great. Unfortunately, right now there is no function in the canvas API that will convert those entities back to their characters, so you'd have to do that yourself.

Q: Can you do anything fancy in canvas, like put dropshadows on text or shapes?

A: Yes! There are lots of fancy things you can do with canvas and dropshadows is certainly one of them. As you'd expect, you create a shadow by setting properties on the context. For instance, to set the blur size of the shadow, you set context.shadowBlur. You can set the position of the shadow with context.shadowOffsetX and context.shadowOffsetY, and the color with context.shadowColor.

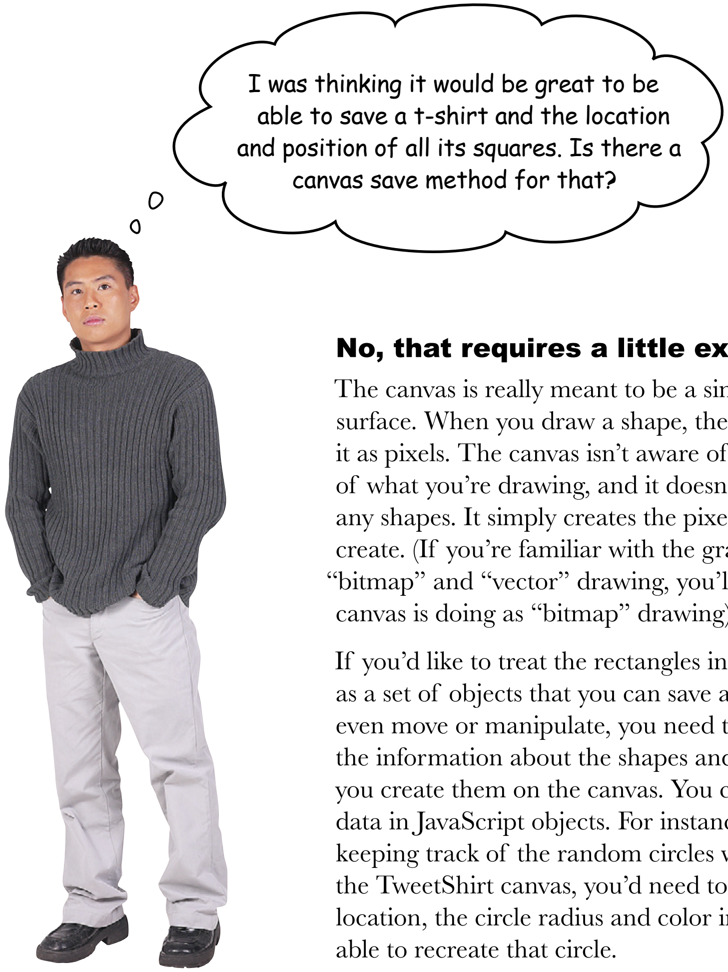
Other things you can do with canvas you might want to check into are things like drawing gradients, rotating shapes, and putting rounded corners on rectangles.

Q: What other interesting things can I do with canvas?

A: A lot! We'll cover a couple more ways to use canvas in later chapters, and you'll definitely want to check out the canvas API for more: http://dev.w3.org/html5/2dcontext/.

Q: Is all this canvas stuff going to work on my mobile device too? Or am I going to have to rewrite it for mobile users?

A: If your mobile device has a modern browser (devices like Android, iPhone and iPad all do), then this will work just fine (the sizing of the page might be off, but the functionality will work. The nice thing about canvas is, because you're drawing with raw pixels, what you draw will look the same everywhere (or, everywhere on browsers that support canvas). Fortunately, modern smart devices like Android, iPhone and iPad all have sophisticated browsers that have most of the functionality of desktop browsers.



No, that requires a little extra work.

The canvas is really meant to be a simple drawing surface. When you draw a shape, the canvas just sees it as pixels. The canvas isn't aware of the specifics of what you're drawing, and it doesn't keep track of any shapes. It simply creates the pixels you ask it to create. (If you're familiar with the graphics terms "bitmap" and "vector" drawing, you'll recognize what canvas is doing as "bitmap" drawing).

If you'd like to treat the rectangles in your canvas as a set of objects that you can save and maybe even move or manipulate, you need to maintain the information about the shapes and paths as you create them on the canvas. You can store this data in JavaScript objects. For instance, if you're keeping track of the random circles we've drawn on the TweetShirt canvas, you'd need to save the x, y location, the circle radius and color in order to be able to recreate that circle.

This sounds like a good project for you.. ;)

Congrats team, you did it! And it even works on my iPad, so this is perfect for customers on the go. I'm thrilled. We're throwing a TweetShirt launch party, so come join us.

The TweetShirt founder also wanted to pass along that she's happy to see the web app works on her iPad and iPhone too! If she's happy, we're happy.

BULLET POINTS

- Canvas is an element you place in your page to create a drawing space.
- The canvas has no default style or content until you provide it (so you won't see it on the page until you draw something in it or add a border with CSS).
- You can have more than one canvas on your page. Of course, you'll need to give each one a unique id to access each using JavaScript.
- To specify the size of the canvas element, use the width and height attributes on the element.
- Everything you put in canvas is drawn using JavaScript.
- To draw on the canvas, you first need to create a context. Currently, a 2D context is your only option, although other context types may exist in the future.
- A context is needed to draw in the canvas because it provides a specific kind of interface (e.g., 2D versus 3D). You'll be able to choose from more than one kind of interface to draw on a canvas.
- You access the canvas by using context properties and methods.
- To draw a rectangle in the canvas, use the context.fillRect method. This creates a rectangle filled with color.
- To create a rectangle outline, use strokeRect instead of fillRect.
- Use fillStyle and strokeStyle to change the default fill and stroke color, which is black.
- You can specify colors using the same format as you use with CSS (e.g., "black", "#000000", "rgb(0, 0, 0)". Remember to put quotes around the value of the fillStyle.
- There is no fillCircle method. To draw a circle in canvas, you need to draw an arc.
- To create arbitrary shapes or arcs, you first need to create a path.
- A path is a invisible line or shape you create that defines a line or area on the canvas. You can't see the path until you stroke or fill it.
- To create a triangle, create a path using beginPath, then use moveTo and lineTo to draw the path. Use closePath to join two points on the path.
- To draw a circle, create an arc that is 360 degrees. Your start angle is 0, and your end angle is 360 degrees.
- Angles are specified in canvas using radians, not degrees, so you need to convert from degrees to radians to specify your start and end angles.
- 360 degrees = 2Pi radians.
- To draw text in canvas, use the fillText method.
- When you draw text in canvas, you need to specify the position, style, and other properties using context properties.
- When you set a context property, it applies to all the drawing that follows until you change the property again. For example, changing the fillStyle will affect the color of shapes and text you draw after setting the fillStyle.
- Add an image to your canvas with the drawImage method.
- To add an image, you first need to create an image object and make sure it's completely loaded.
- Drawing on canvas is like doing "bitmap" drawing in graphics programs.

THE WEBVILLE INQUIRER

We have the scoop, <canvas> and <video> are an item after all!

Webville—You'll hear it here first

After an exclusive interview we can report that <canvas> and <video> have been doing more than just sharing the same web pages. Yeah, you got it... let's just say they've been mixing their content together.

By Troy Armstrong
INQUIRER STAFF WRITER

<Video> says, "It's true, we've formed a tight relationship. You see, I'm a pretty simple guy; I know how to display video, and I do that very well. But that's pretty much all I do. With <canvas>, everything has changed. I'm dressing up in custom controls, I'm filtering my video content, I'm displaying multiple videos at once."

We asked <canvas> to comment. Is she the woman behind the <video> tag? <Canvas> told us, "Well, <video> does very well on his own, you know decoding all those video codecs, maintaining his frames-per-second, all that, it's a very big job, and I could never do it. But with me he has a way to escape his normal, dare I say, "boring" way of playing back video. I give him a means to explore all kinds of creative possibilities of mixing video into the Web experience."

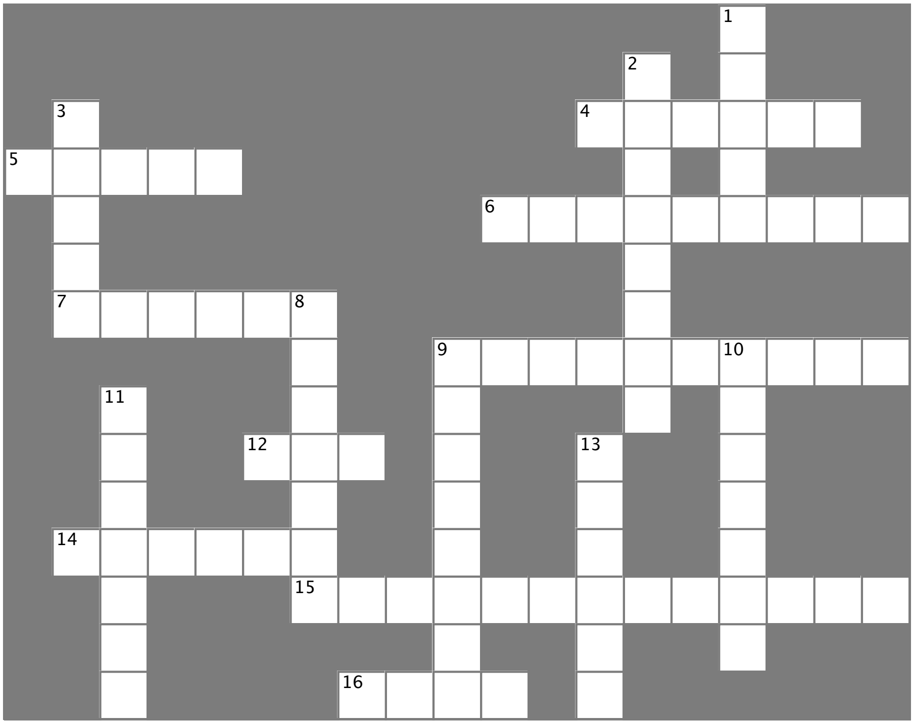
Well, who would have guessed? I guess we've got some interesting things in store ahead from the partnership that is <canvas>+<video>!

The fallout from this revelation can be expected to continue well into the video chapter, when the budding relationship will be exposed to public scrutiny.

Local resident Heidi Musgrove was shocked to learn the truth about the two elements.

HTML5cross

We're looking forward to checking out the <canvas> and <video> scoop in the next chapter. In the meantime, cement your new canvas knowledge with a quick crossword and perhaps a cup of tea.



Across

4. Everything on the canvas is _________ .
5. We _______ aligned the "and all I got was this lousy t-shirt!" text.
6. The property we set to fill a shape with a color.
7. You can tell when something has finished loading using an ________ handler.
9. The non-existent context method Jim tried to use to create circles.
12. Draw a circle with an _____ .
14. How we make the path of a shape visible.
15. Want to know which option is selected? You might need this property.
16. An invisible line you create to draw a shape.

Down

1. Best place for a good tweet.
2. This context method creates a rectangle.
3. Canvas and _______ go well together.
8. There are 360 _____ in a circle.
9. Use this method to draw text on the canvas.
10. We think in degrees, canvas thinks in _________ .
11. An object with methods and properties to draw on a canvas.
13. To move your path pencil to point 100, 100, use ________ (100, 100);

BE the Browser Solution

Now that you have an interface, execute these JavaScript statements and write in the value for each interface element.

Assume you've used the interface to pick the values for your t-shirt.

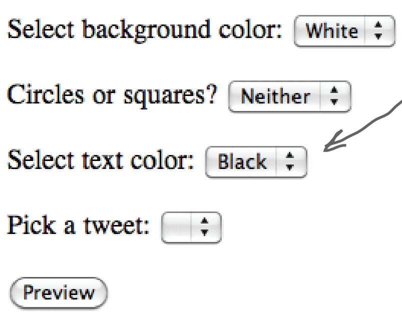
```
var selectObj = document.getElementById("backgroundColor");
var index = selectObj.selectedIndex;
var bgColor = selectObj[index].value;          white

var selectObj = document.getElementById("shape");
var index = selectObj.selectedIndex;
var shape = selectObj[index].value;            circles

var selectObj = document.getElementById("foregroundColor");
var index = selectObj.selectedIndex;
var fgColor = selectObj[index].value;          black
```

Notice that for each menu option value, we get the select element the option is contained in, find the selected option with the selectedIndex property, and the get the value of the selected option.

Remember that the value of the option may be different than the text you see in the controls; in our case, it's just the case of the first letters of the text.

Look at the HTML again to see the values of the options if you need to.

Here are the values we picked in the TweetShirt interface to create the answers above.



Pseudo-code Magnets Solution

Use your pseudo-magical coding powers to arrange the pseudo code below. We need to write the pseudo-code for the drawSquare function. This function takes a canvas and context and draws one randomly sized square on the canvas. Here's our solution.

We did this one for you.

```
function drawSquare ( canvas , context ){
    calculate a random width
    for the square

    calculate a random x position for the
    square inside the canvas

    calculate a random y position for the
    square inside the canvas

    set the fillStyle to "lightblue"

    draw a square at position x,
    y with width w
}
```

Your magnets go here!

Sharpen your pencil Solution

To make sure we see only new squares in the canvas each time we click preview, we need to fill the background of the canvas with the background color the user has selected from the "backgroundColor" select menu. First, let's implement a function to fill the canvas with that color. Fill in the blanks below to complete the code. Here's our solution.

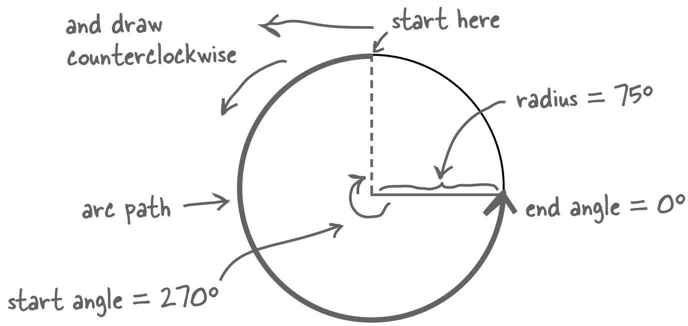
```
function fillBackgroundColor(canvas, context) {
    var selectObj = document.getElementById("backgroundColor");
    var index = selectObj.selectedIndex;
    var bgColor = selectObj.options[index].value;
    context.fillStyle = bgColor;
    context.fillRect(0, 0, canvas.width, canvas.height);
}
```

All we're doing to create a background color is drawing a rectangle that fills the entire canvas with a color.

BE the Browser Solution

Interpret the call to the arc method and sketch out all the values on the circle, including the path the method creates.

```
context.arc(100, 100, 75, degreesToRadians(270), 0, true);
```



Exercise Solution

So you have a path! Now what?

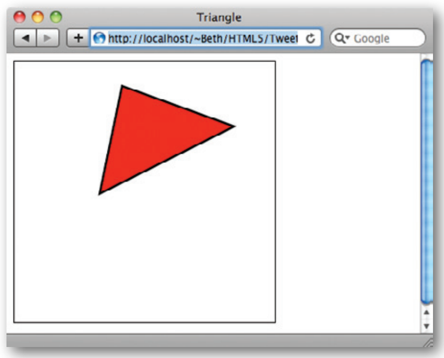
You use the path to draw lines and fill in your shape with color, of course! Go ahead and create a simple HTML5 page with a canvas element and type in all the code so far. Give it a test run, too.

```
context.beginPath();
context.moveTo(100, 150);
context.lineTo(250, 75);
context.lineTo(125, 30);
context.closePath();
```

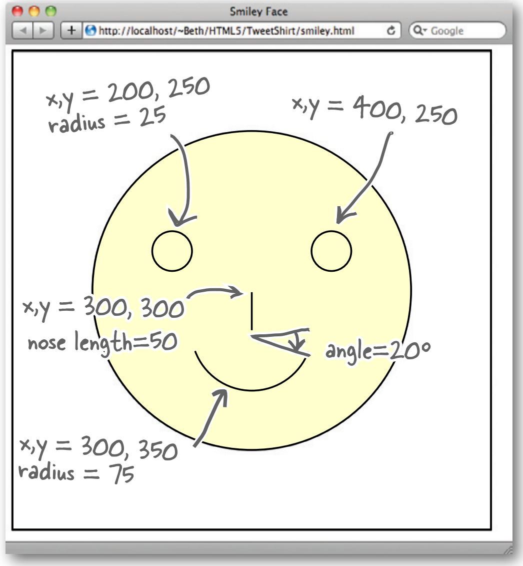
Here's the code so far.

`context.lineWidth = 5;`	Set the width of line to draw over the path.
`context.stroke();`	Draw over the path with the line.
`context.fillStyle = "red";`	Set the color to fill the triangle with to red.
`context.fill();`	Fill the triangle with red.

When we load our triangle page, here's what we get (we made a 300 x 300 canvas to draw in).



Intermission Solution



Time to practice your new arc and path drawing skills to create a smiley face. Fill in the blanks below with the code you need to complete the smiley face. We've given you some hints about where the eyes, nose and mouth should go in the diagram.

Here's our solution:

```
function drawSmileyFace() {
    var canvas = document.getElementById("smiley");
    var context = canvas.getContext("2d");

    context.beginPath();
    context.arc(300, 300, 200, 0, degreesToRadians(360), true);
    context.fillStyle = "#ffffcc";
    context.fill();
    context.stroke();

    context.beginPath();
    context.arc(200, 250, 25, 0, degreesToRadians(360), true);
    context.stroke();

    context.beginPath();
    context.arc(400, 250, 25, 0, degreesToRadians(360), true);
    context.stroke();

    context.beginPath();
    context.moveTo(300, 300);
    context.lineTo(300, 350);
    context.stroke();

    context.beginPath();
    context.arc(300, 350, 75, degreesToRadians(20), degreesToRadians(160), false);
    context.stroke();
}
```

The face circle. We did this one for you. Notice we filled it with yellow.

This is the left eye. The center of the circle is at x=200, y=250, the radius is 25, the starting angle is 0, and the ending angle is Math.PI * 2 radians (360 degrees). We stroke the path so we get the outline of the circle (but no fill).

This is the right eye. Just like the left eye, except it's at x=400. We use counterclockwise (true) for the direction (it doesn't matter for a complete circle).

This is the nose. We use moveTo(300,300) to move the pen to x=300, y=300 to start the line. Then we use lineTo(300,350) because the nose is 50 long. Then we stroke the path.

To get a more realistic smile, we start and end the edge of the mouth at 20 degrees below the x-axis. That means the starting angle is 20°, and the ending angle is 160°.

The direction is clockwise (false) because we want the mouth in a smile. (Remember, the starting point is to the right of the mouth center.)

Code Magnets Solution

It's time for your first experiment with canvas text. Below we've started the code for drawText, the method we're going to call to draw all the text onto the preview canvas. See if you can finish up the code to draw "I saw this tweet" and "and all I got was this lousy t-shirt!" on the canvas, we'll save drawing the actual tweet for later. Here's our solution.

```
function drawText(canvas, context) {
    var selectObj = document.getElementById("foregroundColor");
    var index = selectObj.selectedIndex;
    var fgColor = selectObj[index].value;
    context.fillStyle = fgColor;
    context.font = "bold 1em sans-serif";
    context.textAlign = "left";
    context.fillText("I saw this tweet", 20, 40);

    // Get the selected tweet from the tweets menu
    // Draw the tweet

    context.font = "bold 1em sans-serif";
    context.textAlign = "right";
    context.fillText("and all I got was this lousy t-shirt!",
        canvas.width-20, canvas.height-40);
}
```

Hint: This is the x, y position for the "I saw this tweet" text.

For right now, we're just putting comments in where the code to draw the tweet text will be.

Hint: We'll be using an italic serif font for the tweet, but we want this one to be bold Helvetica.

Hint: we want to position the text in the bottom-right corner.

We want to draw this text at 20 from the right side of the canvas, and 40 from the bottom of the canvas, so it balances the top line of text.

Leftover magnets

`fillCircle` `fillRect` `left`

Exercise Solution

See if you can piece together the drawBird function from all the pieces Judy gave us. The drawBird function takes a canvas and context, and draws the bird onto the canvas. You can assume that with this function we're placing "twitterBird.png" at the location x=20, y=120, with a width and height of 70. We've written the method declaration and the first line for you. Here's our solution.

Your code here →

```
function drawBird(canvas, context) {
    var twitterBird = new Image();
    twitterBird.src = "twitterBird.png";
    twitterBird.onload = function() {
        context.drawImage(twitterBird, 20, 120, 70, 70);
    };
}
```

Don't forget to add a call to drawBird in your previewHandler!

HTML5cross Solution

															1 C			
													2 F		H			
	3 V											4 P	I	X	E	L	S	
5 R	I	G	H	T									L		S			
	D									6 F	I	L	L	S	T	Y	L	E
	E												R					
	7 O	N	L	O	A	8 D							E					
						E			9 F	I	L	L	C	I	10 R	C	L	E
		11 C				G			I				T		A			
		O			12 A	R	C		L			13 M			D			
		N				E			L			O			I			
	14 S	T	R	O	K	E			T			V			A			
		E				15 S	E	L	E	C	T	E	D	I	N	D	E	X
		X							X			T			S			
		T					16 P	A	T	H		O						

TweetShirt Easter egg

So, you've made the perfect TweetShirt preview—now what? Well, if you really want to make a t-shirt out of your design, you can! How? Here's a little extra bonus to add to your code—a TweetShirt "easter egg" if you will—that will make an image out of your design, all ready for you to upload to a site that will print an image on a t-shirt for you (there are plenty of them on the Web).

How can we do this? It's simple! We can use the `toDataURL` method of the **canvas** object. Check it out:

We've made a new function, makeImage, to add this functionality.

```
function makeImage() {
    var canvas = document.getElementById("tshirtCanvas");
    canvas.onclick = function () {
        window.location = canvas.toDataURL("image/png");
    };
}
```

We grab the canvas object...

And add an event handler so that when you click on the canvas, it creates the image.

We set the browser window location to the image that's generated, so you'll see a browser page with just the image in it.

We're asking canvas to create a png image of the pixels drawn on the canvas.

Note that png is the only format that must be supported by browsers, so we recommend you use it.

Now, just add a call to `makeImage` in the window onload function and your canvas is now enabled to make an image when you click on it. Give it a try. And let us know if you make a tshirt!

```
window.onload = function() {
    var button = document.getElementById("previewButton");
    button.onclick = previewHandler;
    makeImage();
}
```

Call makeImage to add the click event handler to the canvas, and your easter egg is complete.

Watch it!

Some browsers won't let you grab an image from the canvas if you're running the code from file://.

Run this code from localhost:// or a hosted server if you want it to work across all browsers.

8 not your father's tv

Video

...with special guest star "Canvas"

We don't need no plug-in. After all, video is now a first-class member of the HTML family—just throw a <video> element in your page and you've got instant video, even across most devices. But video is *far more* than *just an element*, it's also a JavaScript API that allows us to control playback, create our own custom video interfaces and integrate video with the rest of HTML in totally new ways. Speaking of *integration*... remember there's that *video and canvas connection* we've been talking about—you're going to see that putting video and canvas together gives us a powerful new way to *process video* in real time. In this chapter we're going to start by getting video up and running in a page and then we'll put the JavaScript API through its paces. Come on, you're going to be amazed what you can do with a little markup, JavaScript and video & canvas.

Meet Webville TV

Webville TV—all the content you've been waiting for, like *Destination Earth*, *The Attack of the 50' Woman*, *The Thing*, *The Blob*, and it wouldn't be beyond us to throw in a few '50s educational films. What else would you expect in Webville? But that's just the content, on the technology side would you expect anything less than HTML5 video?

Of course, that's just the vision, we have to build Webville TV if we want it to be a reality. Over the next few pages we're going to build Webville TV from the ground up using HTML5 markup, the video element and a little JavaScript here and there.

Webville TV built with 100% HTML5 technology.

Coming to a browser near you soon!

The HTML, let's get it done...

Hey this is Chapter 8, no lollygagging around! Let's jump right in and create some HTML:

```
<!doctype html>
<html lang="en">
<head>
  <title>Webville TV</title>
  <meta charset="utf-8">
  <link rel="stylesheet" href="webvilletv.css">
</head>
<body>
<div id="tv">
   <div id="tvConsole">
      <div id="highlight">
          <img src="images/highlight.png" alt="highlight for tv">
      </div>
      <div id="videoDiv">
          <video controls autoplay src="video/preroll.mp4" width="480" height="360"
                 poster="images/prerollposter.jpg" id="video">
          </video>
      </div>
   </div>
</div>
</body>
</html>
```

Pretty standard HTML5.

Don't forget the CSS file to make it all look nice.

Just a little image to help make it look like a television set.

And here's our <video> element for playing our video. We'll take a closer look in a sec...

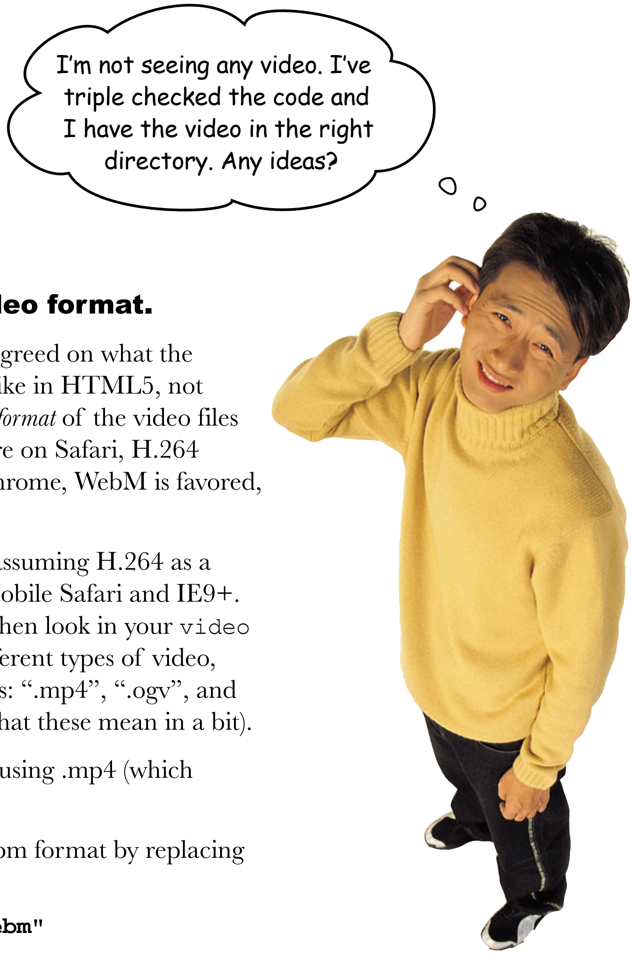
Plug that set in and test it out...

You need to make sure of a few things here: first, make sure you've got the code above typed into a file named `webvilletv.html`; second, make sure you've downloaded the CSS file, and finally, make sure you've also downloaded the video files and placed them in a directory named `video`. After all that, load the page and sit back and watch.

Download everything from http://wickedlysmart.com/hfhtml5

If you're having issues, turn the page!

Here's what we see. Notice if you hover your mouse over the screen you get a set of controls, which you can use to pause, play, set the audio or seek around in the video.



Yes, it's probably the video format.

While the browser makers have agreed on what the <video> element and API look like in HTML5, not everyone can agree on the *actual format* of the video files themselves. For instance if you are on Safari, H.264 format is favored, if you're on Chrome, WebM is favored, and so on.

By the time you read this, these formats could be more widely supported across all browsers. So if your video's working, great. Always check the Web to see the latest on this unfolding topic. And we'll come back for more on this topic shortly.

In the code we just wrote, we're assuming H.264 as a format, which works in Safari, Mobile Safari and IE9+. If you're using another browser then look in your `video` directory and you'll see three different types of video, with three different file extensions: ".mp4", ".ogv", and ".webm" (we'll talk more about what these mean in a bit).

For Safari you should already be using .mp4 (which contains H.264).

For Google Chrome, use the .webm format by replacing your src attribute with:

```
src="video/preroll.webm"
```

If you're using Firefox or Opera, then replace your src attribute with:

```
src="video/preroll.ogv"
```

Give this a try to get you going, and we're coming back to all this in a bit.

And if you're using IE8 or earlier, you're out of luck—wait a sec, this is Chapter 8! How could you still be using IE8 or earlier? Upgrade! But if you need to know how to supply fallback content for your IE8 users, hang on, we're getting to that.

How does the video element work?

At this point you've got a video up and playing on your page, but before we move on, let's step back and look at that video element we used in our markup:

If present, the controls attribute causes the player to supply controls for controlling the video and audio playback.

```
<video controls
       autoplay
       src="video/preroll.mp4"
       width="480" height="360"
       poster="images/prerollposter.jpg"
       id="video">
</video>
```

The autoplay attribute causes the video to start playback upon page load.

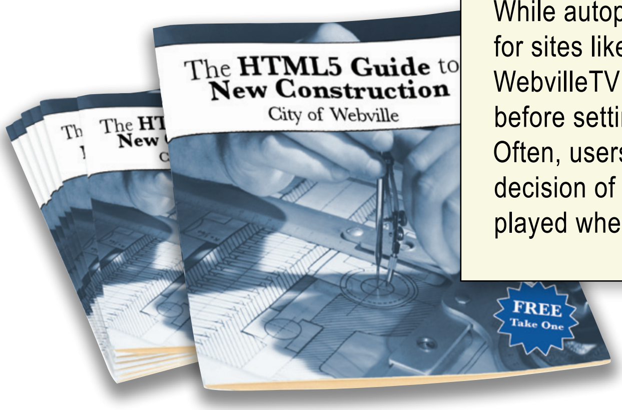
The source location of the video.

The width and height of the video in the page.

A poster image to show when the movie is not playing.

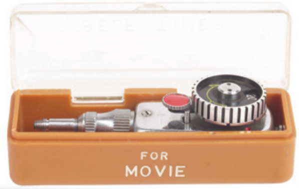
An id for the video element so that we can access it later from JavaScript.

Another handy tip from the HTML5 Guide from the City of Webville.



Good Video Etiquette: The autoplay property

While autoplay may be the best thing for sites like YouTube and Vimeo (or WebvilleTV for that matter), think twice before setting it in your <video> tag. Often, users want to participate in the decision of whether or not video is played when they load your page.

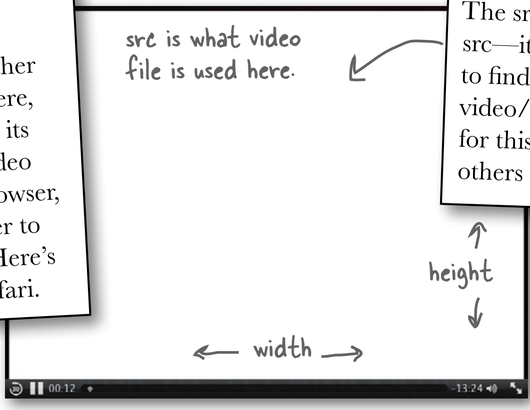


Closely inspecting the video attributes...

Let's look more closely at some of the more important video attributes:

controls

The controls attribute is a **boolean** attribute. It's either there or it's not. If it is there, then the browser will add its built-in controls to the video display. This varies by browser, so check out each browser to see what they look like. Here's what they look like in Safari.



The video player

src

The src attribute is just like the <img> element's src—it is a URL that tells the video element where to find the source file. In this case, the source is video/preroll.mp4. (If you downloaded the code for this chapter, you'll find this video and two others in the video directory).

preload

The boolean attribute preload is typically used for fine-grained control over how video loads for optimization purposes. Most of the time, the browser chooses how much video to load, based on things like whether autoplay is set and the user's bandwidth. You can override this by setting preload to none (none of the video is downloaded until the user "plays"), metadata (the video metadata is downloaded, but no video content), and auto to let the browser make the decision.

autoplay

The autoplay boolean attribute tells the browser to start playing the video as soon as it has enough data. For the videos we're demoing with, you'll probably see them start to play almost immediately.

poster

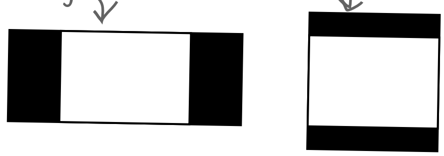
The browser will typically display one frame of the video as a "poster" image to represent the video. If you remove the autoplay attribute you'll see this image displayed before you click play. It's up to the browser to pick which frame to show; often, the browser will just show the first frame of the video... which is often black. If you want to show a specific image, then it's up to you to create an image to display, and specify it by using the poster attribute.

width, height

The width and height attributes set the width and height of the video display area (also known as the "viewport"). If you specify a poster, the poster image will be scaled to the width and height you specify. The video will also be scaled, but will maintain its aspect ratio (e.g., 4:3 or 16:9) so if there's extra room on the sides, or the top and bottom, the video will be letter-boxed or pillar-boxed to fit into the display area size. You should try to match the native dimensions of the video if you want the best performance (so the browser doesn't have to scale in real time).

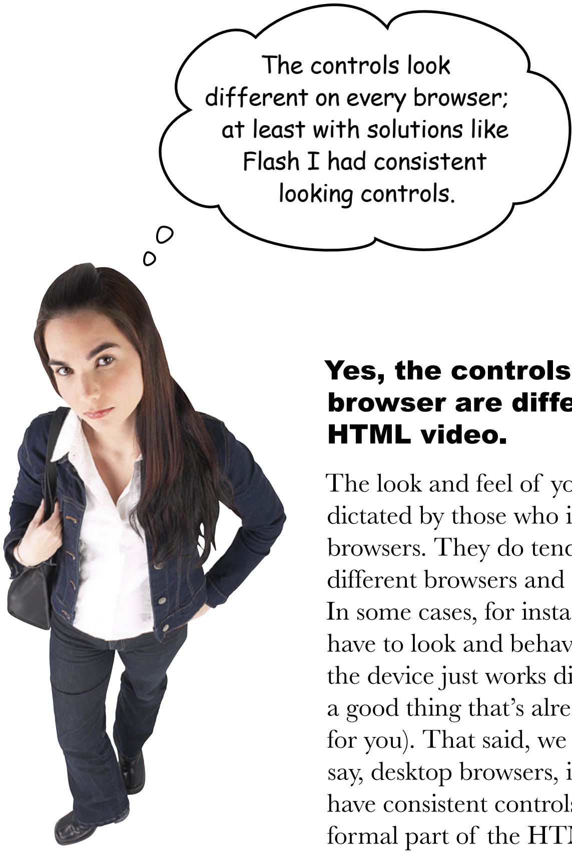
Pillar-boxing

Letter-boxing



loop

Another boolean attribute, loop automatically restarts the video after it finishes playing.



Yes, the controls in each browser are different with HTML video.

The look and feel of your controls is dictated by those who implement the browsers. They do tend to look different in different browsers and operating systems. In some cases, for instance, on a tablet, they have to look and behave differently because the device just works differently (and it's a good thing that's already taken care of for you). That said, we understand; across, say, desktop browsers, it would be nice to have consistent controls, but that isn't a formal part of the HTML5 spec, and in some cases, a method that works on one OS might clash with another operating system's UI guidelines. So, just know that the controls may differ, and if you really feel motivated, you can implement custom controls for your apps.

We'll do this later...

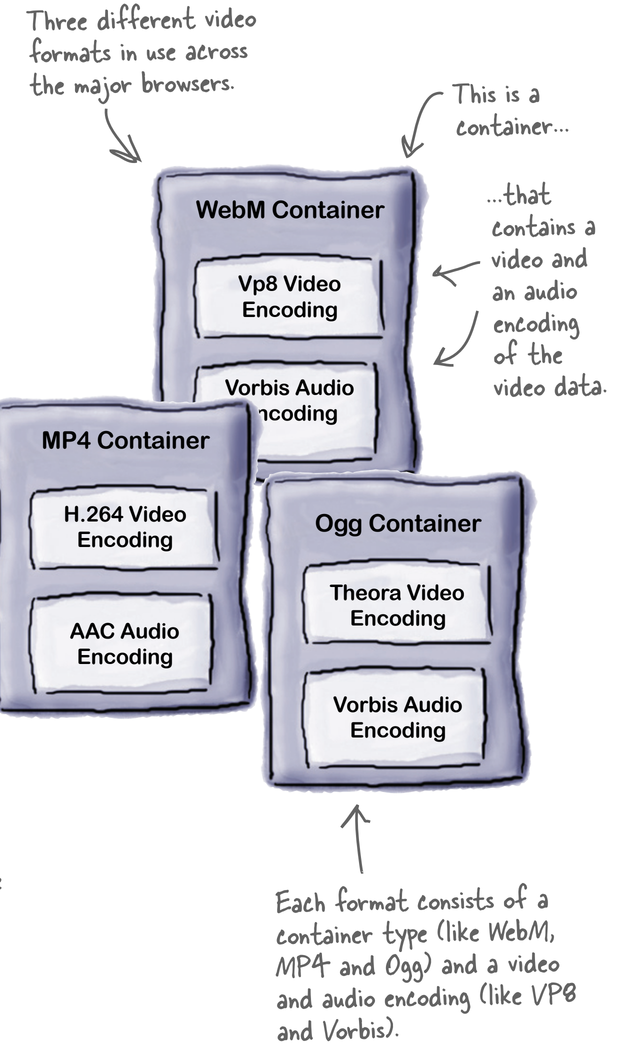
What you need to know about video formats

We wish everything was as neat and tidy as the video element and its attributes, but as it turns out, video formats are a bit of a mess on the Web. What's a video format? Think about it this way: a video file contains two parts, a video part and an audio part, and each part is encoded (to reduce size and to allow it to be played back more efficiently) using a specific encoding type. That encoding, for the most part, is what no one can agree on—some browser makers are enamored with H.264 encodings, others really like VP8, and yet others like the open source alternative, Theora. And to make all this *even more* complicated, the file that holds the video and audio encoding (which is known as a container) has its own format with its own name. So we're really talking buzzword soup here.

Anyway, while it might be a big, happy world if all browser makers agreed on a single format to use across the Web, well, that just doesn't seem to be in the cards for a number of technical, political, and philosophical reasons. But rather than open that debate here, we're just going to make sure you're reasonably educated on the topic so you can make your own decisions about how to support your audience.

Let's take a look at the popular encodings out there; right now there are three contenders trying to rule the (Web) world...

Your mileage may vary by the time you read this book, as favored encodings tend to change over time.



The HTML5 specification allows for any video format. It is the browser implementation that determines what formats are actually supported.

The contenders

The reality is, if you're going to be serving content to a wide spectrum of users you're going to have to supply more than one format. On the other hand, if all you care about is, say, the Apple iPad, you may be able to get away with just one. Today we have three main contenders—let's have a look at them:

MP4 container with H.264 Video and AAC Audio

H.264 is licensed by the MPEG-LA group.

There is more than one kind of H.264; each is known as a "profile."

MP4/H.264 is supported by Safari and IE9+. You may find support in some versions of Chrome.

WebM container with VP8 Video and Vorbis Audio

WebM was designed by Google to work with VP8 encoded videos.

WebM/VP8 is supported by Firefox, Chrome and Opera.

You'll find WebM formatted videos with the .webm extension.

Ogg container with Theora Video and Vorbis Audio

Theora is an open source codec.

Video encoded with Theora is usually contained in an Ogg file, with the .ogv file extension.

Ogg/Theora is supported by Firefox, Chrome and Opera.

H.264 the industry darling, but not the reigning champ...

Theora. The open source alternative.

VP8, the contender, backed by Google, supported by others and coming on strong...

How to juggle all those formats...

So we know it's a messy world with respect to video format, but what to do? Depending on your audience you may decide to provide just one format of your video, or several. In either case, you can use one `<source>` element (not to be confused with the `src` *attribute*) per format inside a `<video>` element, to provide a set of videos, each with its own format, and let the browser pick the first one it supports. Like this:

Notice we're removing the src attribute from the <video> tag...

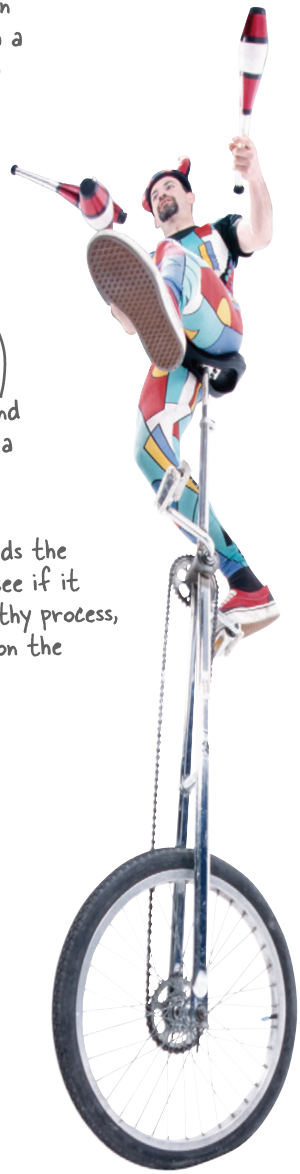
... and adding three source tags each with their own src attribute, each with a version of the video in a different format.

```
<video src="video/preroll.mp4" id="video"
       poster="video/prerollposter.jpg" controls
       width="480" height="360">
    <source src="video/preroll.mp4">
    <source src="video/preroll.webm">
    <source src="video/preroll.ogv">
    <p>Sorry, your browser doesn't support the video element</p>
</video>
```

This is what the browser shows if it doesn't support video.

The browser starts at the top and work its way down until it finds a format it can play.

For each source the browser loads the metadata of the video file to see if it can play it (which can be a lengthy process, although we can make it easier on the browser... see the next page).



BULLET POINTS

- The **container** is the file format that's used to package up the video, audio and metadata information. Common container formats include: MP4, WebM, Ogg and Flash Video.
- The **codec** is the software used to encode and decode a specific encoding of video or audio. Popular web codecs include: H.264, VP8, Theora, AAC, and Vorbis.
- The browser decides what video it can decode and not all browser makers agree, so if you want to support everyone, you need multiple encodings.

How to be even more specific with your video formats

Telling the browser the location of your source files gives it a selection of different versions to choose from, however the browser has to do some detective work before it can truly determine if a file is playable. You can help your browser even more by giving it more information about the MIME type and (optionally) codecs of your video files:

The file you use in the src is actually a container for the actual video (and audio and some metadata).

The codecs parameter specifies which codecs were used for encoding the video and audio to create the encoded video file.

The video codec.

The audio codec.

```
<source src="video/preroll.ogv" type='video/ogg; codecs="theora, vorbis"'>
```

Type is an optional attribute that is a hint to the browser to help it figure out if it can play this kind of file.

This is the MIME type of the video file. It specifies the container format.

Notice the double quotes on the codecs parameter. This means we need to use single quotes around the type attribute.

We can update our <source> elements to include the type information for all three types of video we have, like this:

```
<video id="video" poster="video/prerollposter.jpg" controls width="480" height="360">
    <source src="video/preroll.mp4" type='video/mp4; codecs="avc1.42E01E, mp4a.40.2"'>
    <source src="video/preroll.webm" type='video/webm; codecs="vp8, vorbis"'>
    <source src="video/preroll.ogv" type='video/ogg; codecs="theora, vorbis"'>
    <p>Sorry, your browser doesn't support the video element</p>
</video>
```

If you don't know the codecs parameters, then you can leave them off and just use the MIME type. It will be a little less efficient, but most of the time, that's okay.

The codecs for mp4 are more complicated than the other two because h.264 supports various "profiles," different encodings for different uses (like high bandwidth vs. low bandwidth). So, to get those right, you'll need to know more details about how your video was encoded.

If and when you do your own video encoding, you'll need to know more about the various options for the type parameters to use in your source element. You can get a lot more information on type parameters at `http://wiki.whatwg.org/wiki/Video_type_parameters`.

there are no Dumb Questions

Q: Is there any hope of getting to one container format or codec type in the next few years? Isn't this why we have standards?

A: There probably won't be one encoding to rule them all anytime soon—as we said earlier, this topic intersects with a whole host of issues including companies wanting to control their own destiny in the video space to a complex set of intellectual property issues. The HTML5 standards committee recognized this and decided not to specify the video format in the HTML5 specification. So, while in principle HTML5 supports (or is at least agnostic to) all of these formats, it is really up to the browser makers to decide what they do and don't support.

Keep an eye on this topic if video is important to you; it will surely be an interesting one to watch over the next few years as this is all sorted out. And, as always, keep in mind what your audience needs and make sure you're doing what you can to support them.

Q: If I want to encode my own video, where do I start?

A: There are a variety of video capture and encoding programs out there, and which one you choose is really going to depend on what kind of video you're capturing and how you want to use the end result. Entire books have been written on video encoding, so be prepared to enter a world of all new acronyms and technology. You can start simple with programs like iMovie or Adobe Premiere Elements, which include the ability to encode your video for the Web. If you're getting into serious video work with Final Cut Pro or Adobe Premiere, these software programs include their own production tools. And, finally, if you are delivering your videos from a Content Delivery Network (CDN), many CDN companies also offer encoding services. So you've got a wide variety of choices depending on your needs.

Q: Can I play my video back fullscreen? I am surprised there isn't a property for this in the API.

A: That functionality hasn't yet been standardized, although you'll find ways to do it with some of the browsers if you search the Web. Some of the browsers supply a fullscreen control (for instance, on tablets) that give the video element this capability. Also note that once you've got a way to go fullscreen, what you can do with the video, other than basic playback, may be limited for security reasons (just as it is with plug-in video solutions today).

Q: What about the volume of my video? Can I use the API to control the volume level?

A: You sure can. The volume property can be set to an floating point value between 0.0 (sound off) to 1.0 (sound the loudest). Just use your video object to set this at any time:

```
video.volume = 0.9;
```

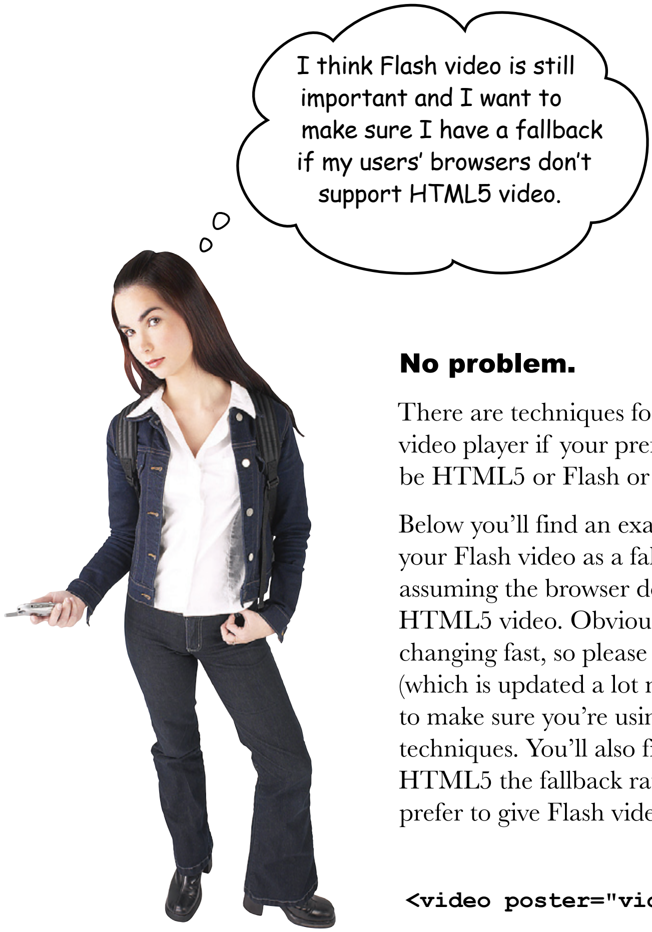
YOUR NEXT MISSION:
VIDEO RECONNAISSANCE

TOP SECRET

GO OUT AND DETERMINE [redacted] THE CURRENT LEVEL OF SUPPORT FOR VIDEO IN EACH BROWSER BELOW (HINT, HERE ARE A FEW SITES THAT KEEP UP WITH SUCH THINGS: HTTP://EN.WIKIPEDIA.ORG/WIKI/HTML5_VIDEO, [redacted] HTTP://CANIUSE.COM/#SEARCH=VIDEO). ASSUME THE LATEST VERSION OF THE BROWSER. FOR EACH BROWSER/FEATURE PUT A CHECKMARK IF IT IS SUPPORTED. UPON YOUR RETURN, REPORT BACK FOR YOUR NEXT ASSIGNMENT!

iOS and Android devices (among others)

Video \ Browser	Safari	Chrome	Firefox	Mobile Webkit	Opera	IE9+	IE8	IE7 or <
H.264								
WebM								
Ogg Theora								



No problem.

There are techniques for falling back to another video player if your preferred one (whether that be HTML5 or Flash or another) isn't supported.

Below you'll find an example of how to insert your Flash video as a fallback for HTML5 video, assuming the browser doesn't know how to play HTML5 video. Obviously this is an area that is changing fast, so please take a look on the Web (which is updated a lot more often than a book) to make sure you're using the latest and greatest techniques. You'll also find ways to make HTML5 the fallback rather than Flash if you prefer to give Flash video priority.

```
<video poster="video.jpg" controls>
        <source src="video.mp4">
        <source src="video.webm">
        <source src="video.ogv">
        <object>...</object>
</video>
```

Insert your <object> element inside the <video> element. If the browser doesn't know about the <video> element, the <object> will be used.

I was told there would be APIs?

As you can see, you can do a lot using markup and the `<video>` element. But the `<video>` element also exposes a rich API that you can use to implement all kinds of interesting video behaviors and experiences. Here's a quick summary of some of the methods, properties and events of the `<video>` element you might be interested in (and check the spec for a comprehensive list):

Call these Methods

play ← plays your video

pause ← pauses your video

load ← loads your video

canPlayType ← helps you determine which video types you can play, programmatically

Use these Properties

videoWidth	loop
videoHeight	muted
currentTime	paused
duration	readyState
ended	seeking
error	volume

These are all properties of the <video> element object. Some you can set (like loop and muted); some are read only (like currentTime and error).

Catch these Events

play	abort
pause	waiting
progress	loadeddata
error	loadedmetadata
timeupdate	volumechange
ended	

And these are all events you can handle if you want by adding event handlers that are called when the event you're listening for occurs.

A little content "programming" on Webville TV

So far we've got one single video up and running on Webville TV. What we'd really like is a programming schedule that serves up a playlist of videos. Let's say we want to do this on Webville TV:

1. Show a little preshow to the audience, you know, the Coke and popcorn ads, the audience etiquette, and so on...

2. Show our first feature, titled *Are you Popular?* Trust us, you'll enjoy it.

3. And then show our featured presentation, *Destination Earth*, presented in full technicolor. Created by the American Petroleum Institute, what on earth could be their message? Watch and find out.

Sharpen your pencil

Now you might be tempted to look at the <video> markup specs to see how you specify a playlist, but for this you're going to need code because the <video> element allows you to specify only one video. If you were on a desert island and had to implement a playlist with only the browser, the <video> element, the src property, the load and play methods, and the ended event, how would you do it (you can use any JavaScript data types you like):

Just a hint: the ended event happens when a video reaches the end and stops playing. Like any event, you can have a handler called when this happens.

No peeking at the answer!!

Sharpen your pencil Solution

Now you might be tempted to look at the <video> markup specs to see how you specify a playlist, but for this you're going to need code because the <video> element allows you to specify only one video. If you were on a desert island and had to implement a playlist with only the browser, the <video> element, the src property, the load and play methods, and the ended event, how would you do it (you can use any JavaScript data types you like)? Here's our solution:

When the page loads we set up a playlist array, start the first video playing, and set up an event handler for when it stops.

Playlist Pseudo-code

Create array of playlist videos

Get video from DOM

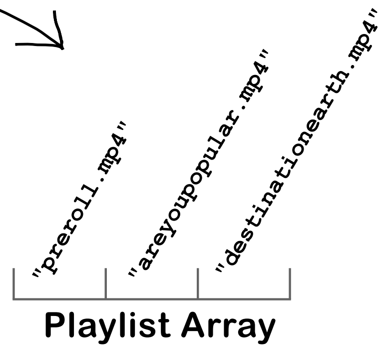
Set event handler on video for "ended" event

Create variable position = 0

Set video source to playlist position 0

Play the video

Here's how we're going to store the playlist, as an array. Each item is a video to play.



Here's our handler to deal with video ending.

Ended Event

Every time a video finishes playing the ended event occurs...

... which calls the ended event handler.

Ended Event Handler Pseudo-code

Increment position by one

Set video to next playlist position

Play the next video

When we get to the end of our playlist we can either stop, or loop around to the first video.

Implementing Webville TV's playlist

Now we're going to use JavaScript and the video API to implement the Webville TV playlist. Let's start by adding a link to a JavaScript file in `webvilletv.html`; just add this into the `<head>` element:

```
<script src="webvilletv.js"></script>
```

And delete this from your existing `<video>` element:

```
<video controls autoplay src="video/preroll.mp4" width="480" height="360"
       poster="images/prerollposter.jpg" id="video">
</video>
```

We're removing the autoplay and src attributes from the <video> tag.

Also remove any <source> elements you might have been experimenting with.

Now, create a new file `webvilletv.js`, and let's define a few global variables and a function that will be called when the page is fully loaded:

First let's define a variable to keep track of which video we're playing; we'll name this position.

```
var position = 0;
var playlist;
var video;
```

And we need a variable to hold the video playlist array.

And also a variable to hold a reference to the video element.

```
window.onload = function() {
    playlist = ["video/preroll.mp4",
                "video/areyoupopular.mp4",
                "video/destinationearth.mp4"];
    video = document.getElementById("video");
    video.addEventListener("ended", nextVideo, false);
```

We'll set up our playlist with three videos.

Grab the video element.

And add a handler for the video ended event. Yes, this looks different than what we're used to—hold on one sec, we'll talk about this on the next page.

```
    video.src = playlist[position];
    video.load();
    video.play();
}
```

Now let's set the src for the first video.

And load the video and play it!

So what's up with that event handler code?

In the past we've always just assigned a handler function to be called when an event occurs to a property (like `onload` or `onclick`) like this:

```
video.onended = nextVideo;
```

However, this time we're going to do things a little differently. Why? Because at the time we're writing this, support for all the event properties on the video object are a little spotty. That's okay; that deficiency is also going to allow us to show you another way of registering for events: `addEventListener`, which is a general method supported by many objects for registering for various events. Here's how it works:

You can use the addEventListener method to add an event handler.

This is the function we're going to call when the event happens.

```
video.addEventListener("ended", nextVideo, false);
```

This is the object on which we're listening for the event.

This is the event we're listening for. Notice we don't put an "on" before the event name like we do with handlers that we set with properties (like onload).

The third parameter controls some advanced methods of getting events if it is set to true. Unless you're writing advanced code you'll always set this to false.

Other than the fact the `addEventListener` method is a little more complicated than just adding a handler by setting the property to a function, it works pretty much the same way. So let's get back to our code!

How to write the "end of video" handler

Now we just need to write the handler for the video's ended event. This handler is going to be called whenever the video player hits the end of the current video file. Here's how we write the `nextVideo` function (add it to `webvilletv.js`):

Note the handler won't be called if the user pauses the video or if the video is looping (which you can do by setting the loop property).

```
function nextVideo() {
    position++;
    if (position >= playlist.length) {
        position = 0;
    }
    video.src = playlist[position];
    video.load();
    video.play();
}
```

First, increment the position in the playlist array.

And if we hit the end of the list, we'll just loop back around by setting the position to zero again.

Now let's set the source of the player to the next video.

And finally, let's load and start the new video playing.

Another test drive...

Can you believe we're ready for a test drive? All we did was use the API to set up a video to play, then we made sure we had an event listener ready to handle the situation when the video ends, which it does by starting the next video in the playlist. Make sure you've got the changes made to your HTML file, type in your new JavaScript code and give it a test drive.

Here's what we see, feel free to scrub ahead in the video to see the video change from one to another without watching the whole show.

It works! But how do we decide which video format to play when we're using code to load the video source?

Good question.

When we were using multiple `source` tags we could count on the browser to sort through one or more video formats and decide if it could play any of them. Now that we're using code we're just giving the video element a single option. So how do we test to see what the browser supports to make sure we supply the best format?

We do that using the `canPlayType` method of the video object. `canPlayType` takes a video format and returns a string that represents how confident the browser is that it can play that type of video. There are three confidence levels: probably, maybe or no confidence. Let's take a closer look and then rework the playlist code to use this.

Are you scratching your head saying "probably? maybe? why doesn't it return true or false?" Us too, but we'll go through what this means in a sec...

How the canPlayType method works

The video object provides a method `canPlayType` that can determine how likely you are to be able to play a video format. The `canPlayType` method takes the same format description you used with the `<source>` tag and returns one of three values: the empty string, "maybe" or "probably". Here's how you call `canPlayType`:

If we pass just the short form of a format then we can only get "" or "maybe" as a result.

```
video.canPlayType("video/ogg")

video.canPlayType('video/ogg; codecs="theora, vorbis"')
```

But if we pass the specific type with a codec, we might then get "", "maybe" or "probably" as an answer.

Empty string if the browser knows it can't play the video.

The string "maybe" if the browser thinks it can possibly play the video.

The string "probably" if the browser is confident it can play the video.

Notice that the browser is only confident beyond "maybe" if you include the codec parameter in the type. Also notice that there's no "I'm absolutely sure" return value. Even if the browser knows it can play a *type* of video, there's still no guarantee it can play the *actual* video; for instance, if the bitrate of the video is too high, then the browser won't be able to decode it.

Bitrate is the number of bits the browser has to process per unit of time to decode the video and display it correctly.

Putting canPlayType to use

We're going to use `canPlayType` to determine which video format to use for Webville TV videos—you already know that we have three versions of each file: MP4, WebM and Ogg, and depending on which browser you're using, some will work and some won't. Let's create a new function that returns the file extension (".mp4", ".webm" or ".ogv") that is appropriate for your browser. We're going to use only the MIME types (`"video/mp4"`, `"video/webm"` and `"video/ogg"`) and not the codecs, so the only possible returns values are "maybe" and the empty string. Here's the code:

```
function getFormatExtension() {
    if (video.canPlayType("video/mp4") != "") {
        return ".mp4";
    } else if (video.canPlayType("video/webm") != "") {
        return ".webm";
    } else if (video.canPlayType("video/ogg") != "") {
        return ".ogv";
    }
}
```

We know we'll only get "maybe" and empty string as answers, so we'll just make sure our matching type doesn't result in an empty string.

We try each of the types, and return the corresponding file extension if the browser says, "Maybe I can support that."

For most use cases, if you don't know the codecs, it's good enough to be "maybe" confident.

Integrating the getFormatExtension function

Now, we need to make some changes to the `window.onload` and `nextVideo` functions to use `getFormatExtension`. First, we'll remove the file extensions from the filenames in the playlist (because we're going to figure those out using `getFormatExtension` instead), and then we'll call `getFormatExtension` where we set the `video.src` property:

```
window.onload = function() {
    playlist = ["video/preroll",
                "video/areyoupopular",
                "video/destinationearth"];
    video = document.getElementById("video");
    video.addEventListener("ended", nextVideo,false);
    video.src = playlist[position] + getFormatExtension();
    video.load();
    video.play();
}
```

Remove the file extensions. We're figuring these out programmatically now.

And concatenate the result of getFormatExtension to the filename for the new video src.

And do the same thing in `nextVideo`:

```
function nextVideo() {
    position++;
    if (position >= playlist.length) {
        position = 0;
    }
    video.src = playlist[position] + getFormatExtension();
    video.load();
    video.play();
}
```

Same thing here; we concatenate the result of getFormatExtension to the video src.

And test drive...

Add the `canPlayType` function and make the changes above, then reload your `webvilletv.html` file. Work? Now your code is figuring out the best format. If you want to know which video the browser chose, try adding an alert to the `window.onload` and `nextVideo` functions; add it at the bottom of each function, after `video.play()`:

```
alert("Playing " + video.currentSrc);
```

Which file did your browser play?

there are no Dumb Questions

Q: If I'm programmatically setting the source of my video, and canPlayType says its a "maybe", but yet the playback fails, how can I deal with that?

A: There are a couple of ways to approach this. One is to catch the error and give the video object another source (we'll talk about catching errors at the end of this chapter). The other is to use the DOM to write multiple source tags into the video object at once (just as if you'd typed them into your markup). That way your video object has several choices and you don't have to write more complex error handling code). We're not going to do that in this chapter, but it's a way of giving your video object multiple choices and doing it through code, not markup, if you have advanced needs.

Watch it!

You may need to install Quicktime to play mp4 video in Safari.

Quicktime often comes installed by default, but if it's not, you can download it from http://www.apple.com/quicktime/download/.

Watch it!

Google Chrome has extra security restrictions.

These security restrictions will prevent you from doing some video+canvas operations if you loaded the web page as a file (i.e., your URL will show file:// rather than http://), like we'll be doing in the rest of this chapter. If you try, the app won't work and you'll get no indication of why.

So, for this chapter, we recommend either using a different browser, or running your own server and running the examples from http://localhost.

Watch it!

Make sure your server is serving video files with the correct MIME type.

Whether you're using your own local server, or running an app using video from a hosted server, you need to make sure that the videos are being served correctly. If they're not, they might not work properly.

If you're on a Mac or Linux using a local server, you're most likely using Apache. You can modify the httpd.conf file (if you have root access) or create a .htaccess file in the directory where your video files are stored, and add the following lines:

```
AddType video/ogg  .ogv
AddType video/mp4  .mp4
AddType video/webm  .webm
```

This tells the server how to serve files with those file extensions.

You can install Apache on Windows, and do the same thing. For IIS servers, we recommend looking in the Microsoft online documentation for "Configuring MIME types in IIS."

At some point we have to treat you like a real developer.

In this book we've (hopefully) helped you every step of the way—we've been there to catch you before you fall and to make sure that, in your code, your i's were dotted and your t's were crossed. But, part of being a real developer is jumping in, reading other peoples' code, making out the forest despite all the trees, and working through the complexity of how it all fits together.

Over the rest of this chapter we're going to start to let you do that. Up next we've got an example that is the closet thing to a real web app we've seen so far, and its got lots of pieces, lots of API use and code that handles lots of real details. Now, we can't step you through every single piece, explaining every nuance like we usually do (or this book will be 1200 pages); nor do we want to, because you also need to acquire the skill of putting all the pieces together, *without us.*

Don't worry, we're still here and we're going to tell you what everything does, but we want you to start to learn how to take code, read it, figure it out, and then augment it and alter it to do *what you want it to do.* So, over the next three chapters, we want you to dig into these examples, study them, and get the code in your head. Really... you're ready!

We need your help!

This just in... we just got the contract to build the **Starring You Video** software for their new video booth. What on earth is that? Oh, just the latest HTML5-enabled video messaging booth—a customer enters an enclosed video booth and shoots their own video message. They can then enhance their video using real movie effects; there's an old-time western sepia filter, a black & white film noir filter, and even an otherwordly sci-fi alien filter. Then the customer can send their message to a friend. We went ahead and commited us all to building the video interface and effects processing system for it.

There's a problem though. The video booths won't be available for another six weeks, and when they arrive the code has to be done. So, in the meantime, we're going to get a partly functional demo unit and a few test video files, and we'll write all our code using those. Then when we're done, the Starring You folks can just point the code to the just-captured real video. And of course, remember that all this has to be done using HTML5.

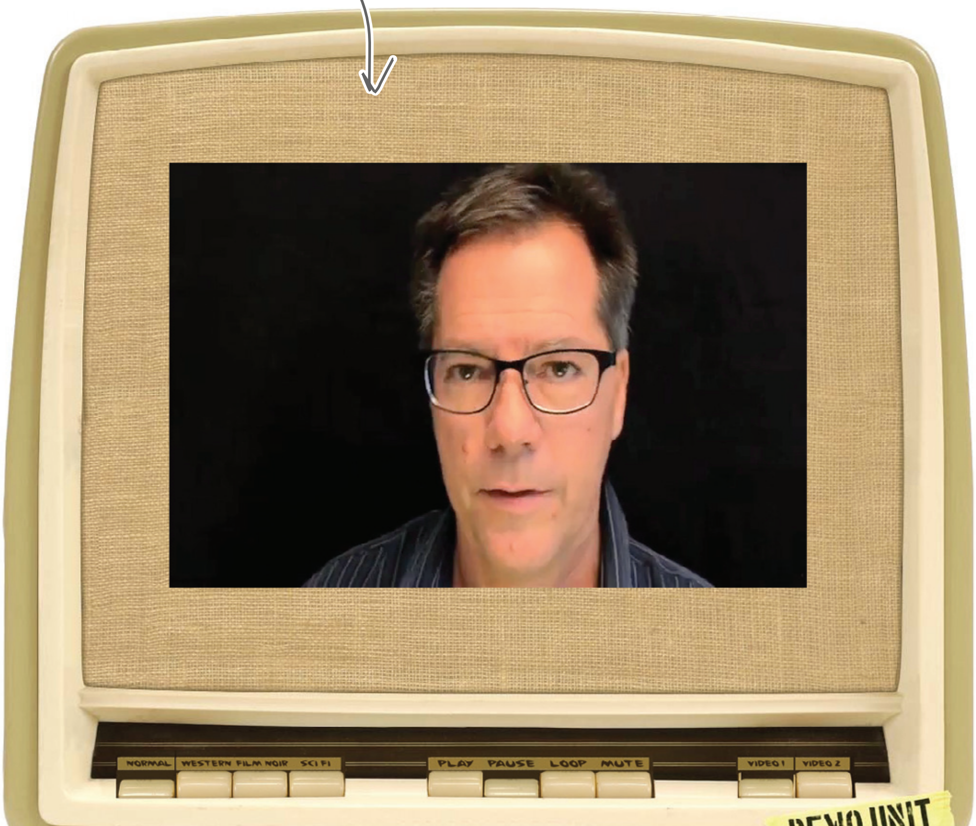
So, we hope you're in, because we signed the contract!

Step in, cut a video, give it a style, and send it to your friends!

Step inside the booth, let's take a look...

Below you'll see our demo unit complete with a user interface. What we've got is a video screen where users will see their video played back. They'll be able to apply a filter like "old-time western" or "sci-fi," see how it looks, and when they're happy, send it off to a friend. We don't have record capability yet, so we've got the test videos to play with. Our first job is going to be to wire everything up so the buttons work, and then write the video filters. Before we get into all that, check out the interface:

Here's the interface of the demo unit. It's got a video player right in the middle for viewing videos.



Apply your favorite effect: old-time western (sepia), film noir (extra dark) or sci-fi (inverted video).

The play, pause, loop and mute controls.

Choose a test video. Our demo unit has two to choose from.

Unpacking the Demo Unit

The demo unit just arrived via next day air and it's time to unpack it. It looks like we've got a functioning unit with some simple HTML markup & JavaScript written so far. Let's have a look at the HTML first (`videobooth.html`). By the way, sit back; we've got a few pages of factory code to look through, and then we'll get cracking on the *real* code.

```
<!doctype html>
```

HTML5, of course.

And all the styling is done for us! Here's the CSS file.

```
<html lang="en">
<head>
    <title>Starring YOU Video Booth</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="videobooth.css">
    <script src="videobooth.js"></script>
```

And here's the JavaScript file, we're going to need to write most of this. We'll take an in-depth look, but it looks they've just written the code to control the buttons on the interface so far...

```
</head>
<body>
<div id="booth">
    <div id="console">
        <div id="videoDiv">
```

Here's the main interface, we've got the console itself, which looks like it is divided into the video display and a dashboard, with three sets of buttons grouped into "effects", "controls" and "videoSelection".

```
            <video id="video" width="720" height="480"></video>
        </div>
```

They've already got a video player installed...good, we're going to need that.

```
        <div id="dashboard">
            <div id="effects">
                <a class="effect" id="normal"></a>
                <a class="effect" id="western"></a>
                <a class="effect" id="noir"></a>
                <a class="effect" id="scifi"></a>
            </div>
```

Here are all the effects.

These are all just HTML anchors. We'll see how we tie into these in a see...

```
            <div id="controls">
                <a class="control" id="play"></a>
                <a class="control" id="pause"></a>
                <a class="control" id="loop"></a>
                <a class="control" id="mute"></a>
            </div>
```

And the player controls.

```
            <div id="videoSelection">
                <a class="videoSelection" id="video1"></a>
                <a class="videoSelection" id="video2"></a>
            </div>
```

And the two demo videos to test with.

```
        </div>
    </div>
</div>
</body>
</html>
```

Inspecting the rest of the factory code

Now let's take a look at all the JavaScript code that shipped from the factory, including the code that sets up the buttons (which we just looked at in the HTML) and the code for each button handler (which, right now, just makes sure the right buttons are depressed). We'll review it all before we start adding our own code.

And now the JavaScript...

So let's crack open the JavaScript (`videobooth.js`). It looks like all the interface buttons work, they just don't do anything interesting, yet. But it's important that we understand how these are set up because the buttons are going to invoke the code we have to write (like, to play a video or to view a video with an effect filter).

Below you'll find the function that is invoked when the page is loaded. For each set of buttons (effects, controls, and the video selection), the code steps through the buttons and assigns click handlers to the anchor links. Let's take a look:

Here's the function that is invoked when the page is fully loaded.

```
window.onload = function() {
    var controlLinks = document.querySelectorAll("a.control");
    for (var i = 0; i < controlLinks.length; i++) {
         controlLinks[i].onclick = handleControl;
    }

    var effectLinks = document.querySelectorAll("a.effect");
    for (var i = 0; i < effectLinks.length; i++) {
        effectLinks[i].onclick = setEffect;
    }

    var videoLinks = document.querySelectorAll("a.videoSelection");
    for (var i = 0; i < videoLinks.length; i++) {
        videoLinks[i].onclick = setVideo;
    }

    pushUnpushButtons("video1", []);
    pushUnpushButtons("normal", []);
}
```

Each for statement loops over the elements of one group of buttons.

The onclick handler for each button in the player controls is set to the handleControl handler.

And the handler for effects is set to setEffect.

And finally the handler for video selection is set to setVideo.

Once we've done all the ground work we use a helper function to visually depress the "video1" button, and the "normal" (no filter) button in the interface.

You haven't seen `document.querySelectorAll` before; it's similar to `document.getElementsByTagName` except that you're selecting elements that match a CSS selector. The method returns an array of element objects that match the CSS selector argument.

```
var elementArray = document.querySelectorAll("selector");
```

Looking at the button handlers

OK, so far the JavaScript code takes care of setting up all the buttons so that if they are clicked on, the appropriate handler is called. Next, let's take a look at the actual handlers, starting with the handler for the player buttons (play, pause, loop and mute), to see what they are doing:

The first thing we do in this handler is see who called us by retrieving the id of the element that invoked the handler.

```
function handleControl(e) {
    var id = e.target.getAttribute("id");

    if (id == "play") {
        pushUnpushButtons("play", ["pause"]);

    } else if (id == "pause") {
        pushUnpushButtons("pause", ["play"]);

    } else if (id == "loop") {
        if (isButtonPushed("loop")) {
            pushUnpushButtons("", ["loop"]);
        } else {
            pushUnpushButtons("loop", []);
        }
    } else if (id == "mute") {
        if (isButtonPushed("mute")) {
            pushUnpushButtons("", ["mute"]);
        } else {
            pushUnpushButtons("mute", []);
        }
    }
}
```

Once we know the id, we know if the element was play, pause, loop or mute.

Depending on which button it was, we alter the interface to reflect the button that was pushed. For instance if pause was pushed then play shouldn't be.

We're using a helper function to make sure the onscreen button states are taken care of, it's called pushUnpushButtons, and it takes a pushed button along with an array of unpushed buttons and updates the interface to reflect that state.

Various buttons have different semantics. For instance play and pause are like true radio buttons (pushing one in pops the other one out), while mute and loop are like toggle buttons.

All this code so far is cosmetic, it just changes the look of the buttons from pressed to depressed. There is no code to do anything real, like play a video. That's what we have to write.

Now that's great and all, but where does *our code* come in? Let's think through this: when a button, like play, is pushed, not only are we going to update the interface (which the code already does), we're also going to add some code that actually *does something*, like make the video start playing. Let's go ahead and look at the other two handlers (for setting the video effects and for setting the test video), and it should be pretty obvious (if it isn't already) where our code is going to go...

The setEffect and setVideo handlers

Let's look at the other two handlers. The `setEffect` handler handles your choice of effect, like no effect (normal), western, film noir or sci-fi. Likewise the `setVideo` handler handles your choice of test video one or two. Here they are:

This works the same as the handleControl handler: we grab the id of the element that called us (the button that was clicked on) and then update the interface accordingly.

```
function setEffect(e) {
    var id = e.target.getAttribute("id");

    if (id == "normal") {
        pushUnpushButtons("normal", ["western", "noir", "scifi"]);

    } else if (id == "western") {
        pushUnpushButtons("western", ["normal", "noir", "scifi"]);

    } else if (id == "noir") {
        pushUnpushButtons("noir", ["normal", "western", "scifi"]);

    } else if (id == "scifi") {
        pushUnpushButtons("scifi", ["normal", "western", "noir"]);

    }
}
```

And here's where our code is going to go.

We'll be adding code to each case to handle implementing the appropriate special effect filter.

The same is true of setVideo; we see which button was pressed and update the interface.

```
function setVideo(e) {
    var id = e.target.getAttribute("id");
    if (id == "video1") {
        pushUnpushButtons("video1", ["video2"]);
    } else if (id == "video2") {
        pushUnpushButtons("video2", ["video1"]);
    }

}
```

We'll also be adding code here to implement switching between the two test videos.

And here are the helper functions

And for the sake of completeness (or if you're on a 11-hour flight to Fiji without Internet access and you really want to type all this in):

And remember, if you don't want to type it in, you can get all the code from http://wickedlysmart.com/hfhtml5.

pushUnpushButtons takes care of button states. The arguments are the ids of a button to push in, and one or more buttons to unpush in an array.

First, check to make sure the id of the button to push is not empty.

Grab the anchor element using that id...

... and get the class attribute.

We "press" the button by adding the "selected" class to the anchor, and ...

... update the background image of the anchor element so it covers up that button with a "button pressed" image. So "pause" uses the "pausepressed.png" image.

```
function pushUnpushButtons(idToPush, idArrayToUnpush) {
    if (idToPush != "") {
        var anchor = document.getElementById(idToPush);
        var theClass = anchor.getAttribute("class");
        if (!theClass.indexOf("selected") >= 0) {
            theClass = theClass + " selected";
            anchor.setAttribute("class", theClass);
            var newImage = "url(images/" + idToPush + "pressed.png)";
            anchor.style.backgroundImage = newImage;
        }
    }
```

To unpush buttons, we loop through the array of ids to unpush, grabbing each anchor...

... make sure the button is really pushed (if it is, it will have a "selected" class)...

... remove "selected" from the class...

... and remove the background image so we see the unpushed button.

```
    for (var i = 0; i < idArrayToUnpush.length; i++) {
        anchor = document.getElementById(idArrayToUnpush[i]);
        theClass = anchor.getAttribute("class");
        if (theClass.indexOf("selected") >= 0) {
            theClass = theClass.replace("selected", "");
            anchor.setAttribute("class", theClass);
            anchor.style.backgroundImage = "";
        }
    }
}
```

isButtonPushed simply checks to see if a button is pushed. It takes the id of an anchor...

... grabs the anchor...

... gets the class of that anchor...

... and returns true if the anchor has the "selected" class.

```
function isButtonPushed(id) {
    var anchor = document.getElementById(id);
    var theClass = anchor.getAttribute("class");
    return (theClass.indexOf("selected") >= 0);
}
```

That new demo machine smell...test drive time!

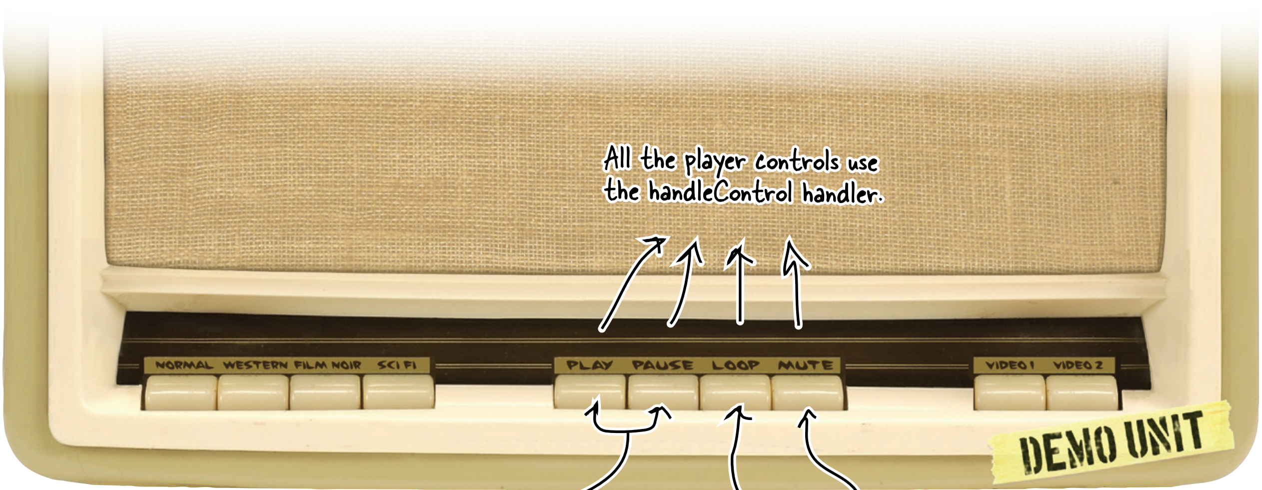
We haven't done much writing code, but we are reading and understanding code, and that can be just as good. So load the `videobooth.html` file into your browser and check out the buttons. Give them a good testing. For extra credit, add some alerts into the handler functions. Get a good feel for how this is working. When you come back, we'll start writing some code to make the buttons work for real.

Give all those buttons a try, notice some are like radio buttons, some are like toggle buttons.

Sharpen your pencil

You'll find the solution just a couple of pages on...

Mark the buttons below with whether they are like toggle buttons (independent) or like radio buttons (pushing one in pops the other ones out). Also annotate each button with its corresponding click handler. We've done a couple for you:



All the player controls use the handleControl handler.

Radio buttons, play and pause can't be selected at the same time.

Loop and Mute are toggle buttons, they can be used independently of any other buttons.

I think I missed something... how did you get from <div>s with anchor tags to having buttons in the interface?

That would be the power of CSS.

It's a shame this book isn't *Head First HTML5 Programming with JavaScript & CSS*, but then it would be 1,400 pages, wouldn't it? Of course, we could be talked into writing an advanced CSS book...

But seriously, this is the power of markup for structure and CSS for presentation (and if that is a new topic to you, check out *Head First HTML with CSS & XHTML*). What we're doing isn't that complex; here it is in a nutshell for the curious:

We set the background image of the console <div> to the booth console (no buttons).

The <video> element is in a <div> which is positioned relative to the console. Then, the <video> element is absolute positioned so it sits in the middle of the console.

We position the dashboard <div> relative to the console and then position the <div>s for each group of buttons relative to the dashboard.

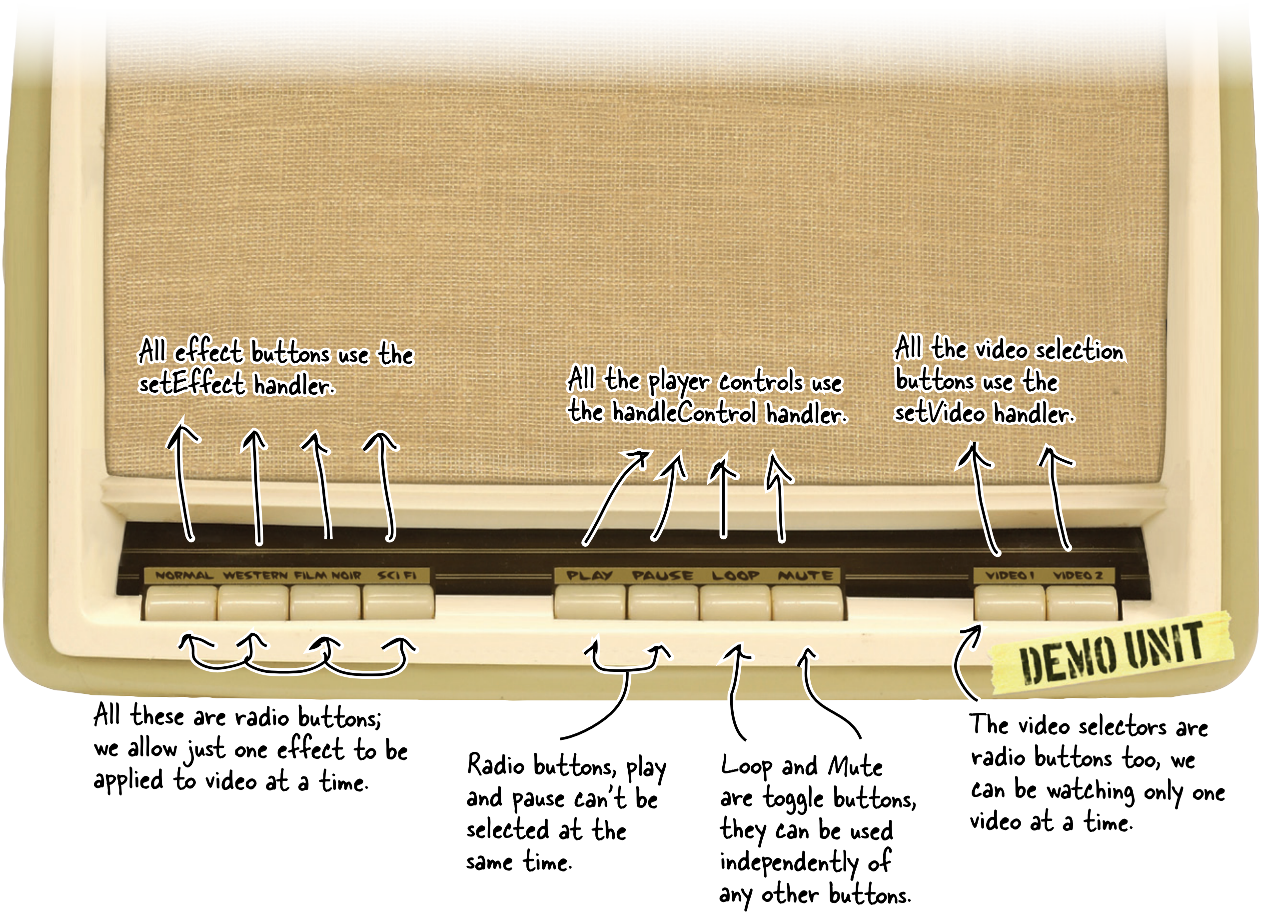
Each button group <div> gets a background image for all the unpushed buttons.

Each "button" anchor is positioned within the <div> for the group, and given a width and height to match the button it corresponds to. When you click on a "button", we give that anchor a background image of a pushed in button to cover up the unpushed button.

Check out the CSS in detail if you want: videobooth.css.

Sharpen your pencil Solution

Mark the buttons below with whether they are like toggle buttons (independent) or like radio buttons (pushing one in pops the other ones out). Also annotate each button with its corresponding click handler. Here's our solution:



Getting our demo videos ready...

Before we implement the button controls, we need video to test them with, and as you can see from the buttons, Starring You Video has sent us two demo videos. Let's go ahead and create an object to hold the two videos and then add some code for our onload handler to set up the source of the video object (just like we did for Webville TV).

We'll create this object to hold the two demo videos. We'll come back and explain more about this shortly.

```
var videos = {video1: "video/demovideo1", video2: "video/demovideo2"};

window.onload = function() {

    var video = document.getElementById("video");
    video.src = videos.video1 + getFormatExtension();
    video.load();

    var controlLinks = document.querySelectorAll("a.control");
    for (var i = 0; i < controlLinks.length; i++) {
        controlLinks[i].onclick = handleControl;
    }
    var effectLinks = document.querySelectorAll("a.effect");
    for (var i = 0; i < effectLinks.length; i++) {
        effectLinks[i].onclick = setEffect;
    }
    var videoLinks = document.querySelectorAll("a.videoSelection");
    for (var i = 0; i < videoLinks.length; i++) {
        videoLinks[i].onclick = setVideo;
    }
    pushUnpushButtons("video1", []);
    pushUnpushButtons("normal", []);
}
```

Here, we're getting the video element, and setting its source to the first video in the array with a playable extension.

Then we go ahead and load the video so if the user clicks play, it's ready to go.

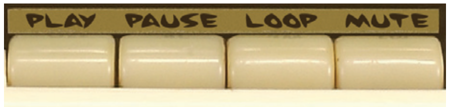
READ THIS CAREFULLY!

Now before we get sloppy, remember the `getFormatExtension` function is in Webville TV, not this code! So open up `webvilletv.js` and copy and paste the function into your video booth code. One other small thing: in the video booth code we aren't keeping a global video object, so add this line to the top of your `getFormatExtension` function to make up for that:

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

Add this line to your getFormatExtension function at the top.

Implementing the video controls



Now we're going to implement all these buttons.

Alright, time to get to those buttons! Now, it's important to point out that for this project, we're going to implement *our own* video controls. That is, rather than use the built-in video controls, we're going to control the experience ourselves—so when the user needs to play, pause or mute the video, or even to loop the playback, they're going to use our custom buttons, not the built-in controls. It also means we're going to do all this *programmatically* through the API. Now, we're not going to go all the way, which would mean implementing our own video scrubber, or perhaps next and previous buttons, because those don't make sense in this application, *but we could if we needed to.* You'll find that just by implementing our small control panel you'll get the idea and be in perfect shape to take it further if you want to.

So, let's get started. How about if we start with the play button and then move to the right (to pause and then to loop and then to mute) from there? So find the `handleControl` handler and add this code:

```
function handleControl(e) {
    var id = e.target.getAttribute("id");
    var video = document.getElementById("video");

    if (id == "play") {
        pushUnpushButtons("play", ["pause"]);
        if (video.ended) {
            video.load();
        }
        video.play();
    } else if (id == "pause") {
        pushUnpushButtons("pause", ["play"]);

    } else if (id == "loop") {
        if (isButtonPushed("loop")) {
            pushUnpushButtons("", ["loop"]);
        } else {
            pushUnpushButtons("loop", []);
        }
    } else if (id == "mute") {
        if (isButtonPushed("mute")) {
            pushUnpushButtons("", ["mute"]);
        } else {
            pushUnpushButtons("mute", []);
        }
    }
}
```

We need a reference to the video object.

This should be pretty simple. If the user has pressed play, then call the play method on the video object.

But we'll warn you, there's one edge case here about to bite us, so let's go ahead and take care of it: If we've played a video, and let that video play through to the end, then to start it playing again, we have to load it again first. We check to make sure the video ran through to the end (and wasn't just paused), because we only want to load again in that case. If it's paused, we can just play without loading.

Implementing the rest of the video controls

Let's knock out the rest of the controls—they're so straightforward they're almost going to write themselves:

```
function handleControl(e) {
    var id = e.target.getAttribute("id");
    var video = document.getElementById("video");

    if (id == "play") {
        pushUnpushButtons("play", ["pause"]);
        video.load();
        video.play();
    } else if (id == "pause") {
        pushUnpushButtons("pause", ["play"]);
        video.pause();

    } else if (id == "loop") {
        if (isButtonPushed("loop")) {
            pushUnpushButtons("", ["loop"]);
        } else {
            pushUnpushButtons("loop", []);
        }
        video.loop = !video.loop;

    } else if (id == "mute") {
        if (isButtonPushed("mute")) {
            pushUnpushButtons("", ["mute"]);
        } else {
            pushUnpushButtons("mute", []);
        }
        video.muted = !video.muted;

    }
}
```

If the user pauses the video, then use the video object's pause method.

For looping we've got a boolean property named loop in the video object. We just set it appropriately...

...and to do that we'll keep you on your toes by using the boolean "!" (not) operator, which just flips the boolean value for us.

And mute works the same way: we just flip the current value of the mute property when the button is pressed.

Another test drive!

Make sure you've got all the code changes typed in. Load `videobooth.html` into your browser and give your control buttons a test. You should see video start playing, be able to pause it, mute it, or even put it in a loop. Of course, you can't select the other demo video yet or add an effect, but we're getting there!

Taking care of a loose end...

There's a little loose end we need to take care of to really make these buttons work like they should. Here's the use case: let's say you're playing a video and you don't have loop selected, and the video plays to completion and stops. As we have things implemented now, the play button will remain in the pressed position. Wouldn't it be better if it popped back up, ready to be pressed again?

Our play button isn't 100% right yet...

Using events we can easily do this. Let's start by adding a listener for the `ended` event. Add this code to the bottom of your `onload` handler:

```
video.addEventListener("ended", endedHandler, false);
```

Set a handler for the "ended" event, which is called when the video playback ends (but NOT when you pause!).

Now let's write the handler, which will be called any time the video playback stops by coming to the end of the video:

```
function endedHandler() {
    pushUnpushButtons("", ["play"]);
}
```

All we need to do is "unpush" the play button. That's it!

And another...

Okay, make the changes, save the code and reload. Now start a video and let it play to its conclusion without the loop button pressed, and at the end you should see the play button pop back out on its own.

The play button should pop right back out after the video finishes playing.

Switching test videos

We already added an object to hold our two test videos, and we've even got two buttons for selecting between them. Each button is assigned the `setVideo` handler. Let's work through writing that now so we can switch between our videos:

Here's our object with the two videos, we're showing this again as a reminder, so you can see how we're going to use it...

```
var videos = {video1: "video/demovideo1", video2: "video/demovideo2"};

function setVideo(e) {
    var id = e.target.getAttribute("id");
    var video = document.getElementById("video");

    if (id == "video1") {
        pushUnpushButtons("video1", ["video2"]);
    } else if (id == "video2") {
        pushUnpushButtons("video2", ["video1"]);
    }
    video.src = videos[id] + getFormatExtension();
    video.load();
    video.play();

    pushUnpushButtons("play", ["pause"]);
}
```

And here's the handler again.

Again, we need a reference to the video object.

Then we still update the buttons in the same way we were, no changes there.

Then we use the source id of the button (the id attribute of the anchor) to grab the correct video filename, and add on our browser-aware extension. Notice we're using the [ ] notation with our videos object, using the id string as the property name.

Once we have the correct video path and filename, we load and play the video.

And we make sure the play button is pushed in because we start the video playing when the user clicks on a new video selection.

Switch drivers and test drive!

Make these changes to your `setVideo` function and then load your page again. You should now be able to easily switch between video sources.

HTML5 Confidential

This week's interview:
Confessions of the Video Element

Head First: Welcome, Video. I'm going to jump right to the topic on everyone's minds, and that would be YOU and the Canvas element.

Video: Meaning?

Head First: Allegedly nights out on the town, early morning breakfasts together. Need I say more?

Video: What is there to say, Canvas and I have a great relationship. She displays her content in a very, let's say, visually appealing manner, and I'm a video workhorse. I crunch through codecs and get that video content to the browser.

Head First: Well, that's not what I was getting at, but I'll go with it. Okay, she's great at 2D display, you're great at video display. So what? What's the real connection?

Video: Like any relationship, when you put two things together and get out more than the sum of the parts, that's when you have something special.

Head First: Okay, well, can you put that in more concrete terms?

Video: It's a simple concept. If you want to do anything other than just basic video playback—that is, if you want to do any processing on your video, or custom overlays or display mulitple videos at once—then you want to use Canvas.

Head First: That all sounds great, but video requires heavy-duty processing, I mean that's a lot of data coming through. How is JavaScript, a scripting language, going to do anything real? Writing JavaScript code isn't like writing in a native language.

Video: Oh you'd be surprised... have you looked at the latest benchmarks on JavaScript? It's already fast, and getting faster every day. The industry's brightest virtual machine jockeys are working the problem and kicking butt.

Head First: Yeah, but video? Really?

Video: Really.

Head First: Can you give us some examples of things you can do with JavaScript, canvas and video?

Video: Sure, you can process video in real time, inspect the video's characteristics, grab data from video frames, and alter the video data by, say... rotating it, scaling it or even changing the pixels.

Head First: Can you walk us through how you might do that in code?

Video: Uh, I'll have to get back to you on that, just got a call from Canvas... gotta run...

It's time for special effects

Isn't it about time we add those movie effects? What we want to do is take our original video and be able to apply effects, like film noir, western, and even an otherworldly sci-fi effect. But if you look at the video API, you won't find any effects methods there, or any way to add them directly. So how are we going to add those effects?

Take a little time to think through how we might add effects to our video. Don't worry that you don't know how to process video yet, just think through the high level design.

We want to take our original video and be able to apply film noir, western, and sci-fi effects.

Starring You Video

Engineering Notes...

Use this engineering note to draw a picture, label it, or write out pseudo-code for any code for your video effects. Think of this as a warm up, just to get your brain going...

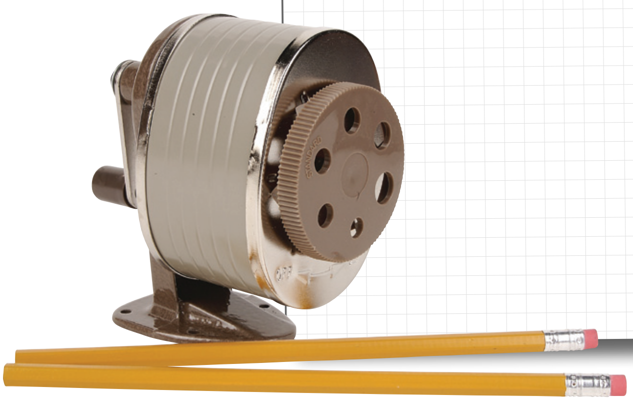
How will you get your hands on the pixels that make up each frame of the video?

Once you've got the pixels, how do you process them to apply the effect?

Say you were to write a function to implement each effect, what would it look like?

How can you display the video once you've processed all its pixels to apply the effect?

Your ideas here.



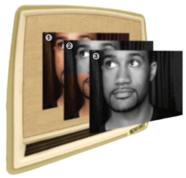
The FX plan

We don't know exactly how to implement the effects yet, but here's a high level plan of attack:

1. We know we've still got to hook up those buttons that control the effects. So we're going to do that first.

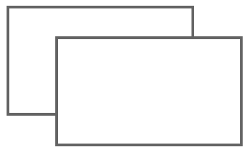
The buttons still need to be wired up.

2. We're going to learn a little about video processing and check out the "scratch buffer" technique for adding our effects.



The scratch buffer, that looks interesting...

3. We're going to implement the scratch buffer, which is going to give us a chance to see video and canvas together in action.



Implement the scratch buffer, using canvas (believe it or not)!

4. We're going to implement one function for each effect: western, film noir, and sci-fi.

```
function noir(pos, r, g, b, data) {
   ...
}
```

5. Finally, we're going to put it all together and test!

We'll display the altered pixels in a, you guessed it, canvas.

BRAIN POWER

Now you know we're going to implement a function that will handle each effect. Let's take film noir, for example. How are you going to take a color pixel from the video and make it black and white? Hint: every pixel has three components: red, green, and blue. If we could get our hands on those pieces, what could we do?

Time to get those effects buttons working

Alright, the easy part first: we're going to get those effects buttons wired up and working. We'll start by creating a global variable named `effectFunction`. This variable is going to hold a function that can take data from the video, and apply a filter to it. That is, depending on which effect we want, the `effectFunction` variable will hold a function that knows how to process the video data and make it black and white, or sepia, or inverted for sci-fi. So add this global variable to the top of your file:

```
var effectFunction = null;
```

Now we're going to set this variable anytime an effects button is clicked on. For now, we'll use function names like `western`, `noir` and `scifi`, and we'll write these functions in just a bit.

Here's our setEffect handler again. Remember this is called whenever the user clicks on a effect button.

For each button press we set the effectFunction variable to the appropriate function (all of which we still need to write).

```
function setEffect(e) {
    var id = e.target.getAttribute("id");

    if (id == "normal") {
        pushUnpushButtons("normal", ["western", "noir", "scifi"]);
        effectFunction = null;
    } else if (id == "western") {
        pushUnpushButtons("western", ["normal", "noir", "scifi"]);
        effectFunction = western;
    } else if (id == "noir") {
        pushUnpushButtons("noir", ["normal", "western", "scifi"]);
        effectFunction = noir;
    } else if (id == "scifi") {
        pushUnpushButtons("scifi", ["normal", "western", "noir"]);
        effectFunction = scifi;
    }
}
```

If the effect is no effect, or normal, we just use null as the value.

Otherwise we set effectFunction to an appropriately named function that will do the work of applying the effect.

We still need to write these effects functions. So, let's see how we process video so we can apply effects to it!

Okay, with that out of the way, we're going to learn about that "scratch buffer" and then come back and see how these functions fit in, as well as how to write them!

How video processing works

What we've done so far is given ourselves a way to assign a function to the `effectsFunction` global variable as a result of clicking on the effects buttons in the interface. For now, just take that knowledge and tuck it in the back of your brain for a little while, because we've got to work through how we're actually going to take video and process it in real time to add an effect. To do that we need to get our hands on the video's pixels, alter those pixels to achieve our desired effect, and then somehow get them back on the screen.

Now, does the video API offer some special way to process video before it is displayed? Nope. But it does give us a way to get the pixels, so we just need a way to process and display them. Wait, pixels? Display? Remember Chapter 7? The **canvas**! Ah, that's right, we did mention something about the "special relationship" that the video element and canvas have. So, let's walk through one of the ways the video and canvas elements can work together:

The details of the scoop, finally revealed!

1. The video player decodes and plays the video behind the scenes.

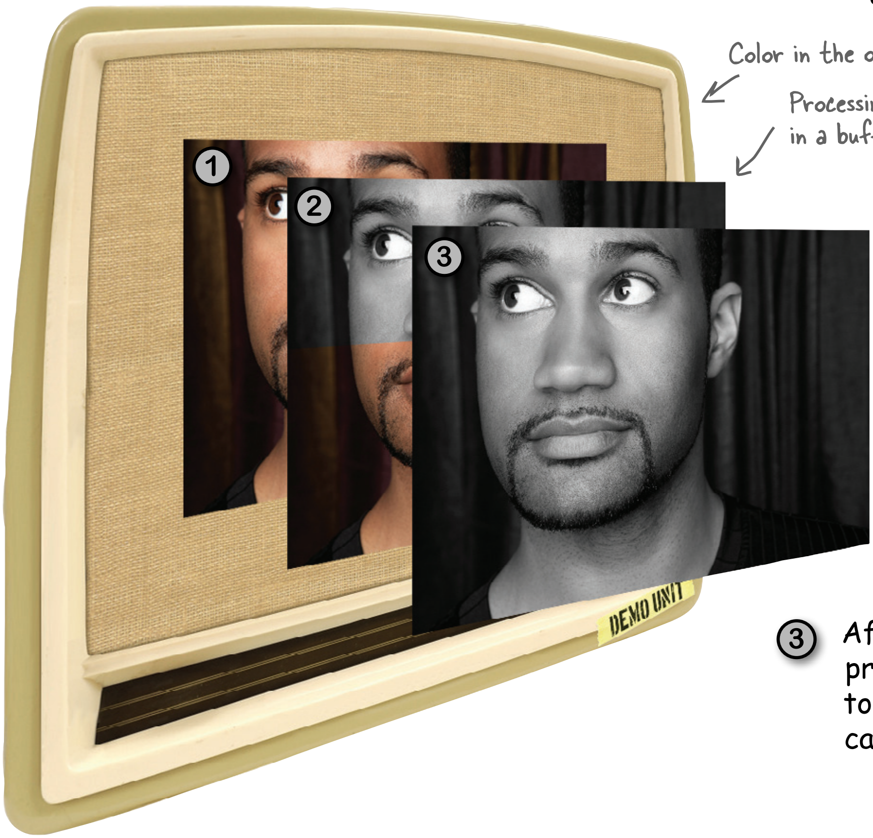
2. Video is copied frame by frame into a (hidden) buffer canvas and processed.

Color in the original video.

Processing color to, say, B&W, in a buffer canvas.

Then we copy the processed frame from the buffer to the display canvas.

In a nutshell, we're taking each frame of video and changing it from color to B&W, and then displaying it.



3. After a frame is processed, it is copied to another display canvas to be viewed.

How to process video using a scratch buffer

Behind the Scenes

Now, you might ask why we're using two canvases to process and display the video. Why not just find a way to process the video as it is decoded?

The method we're using here is a proven technique for minimizing visual glitches during intensive video and image processing: it's known as using a "scratch buffer." By processing a frame of video in a buffer and then copying it all in one fell swoop to the display canvas, we minimize visual issues.

Let's step through how our scratch buffer implementation is going to work.

1. The browser decodes the video into a series of frames. Each frame is a rectangle of pixels with a snapshot of the video at a given point in time.

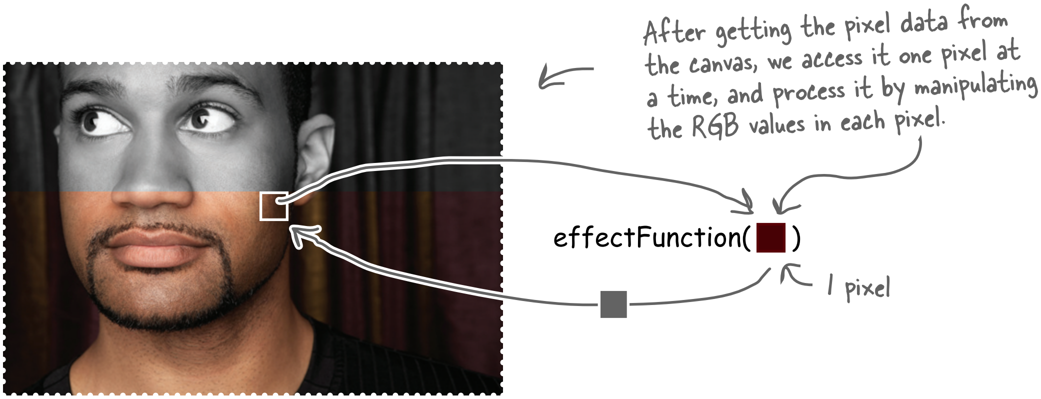
One frame of video.

2. As each frame is decoded we copy it into the canvas that is acting as a scratch buffer.

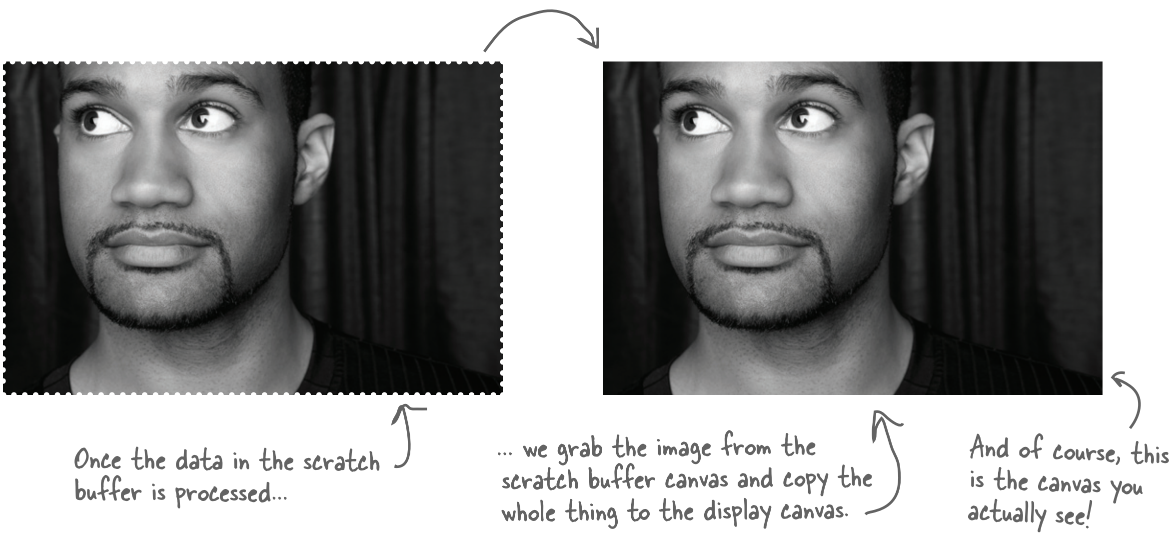
We copy the whole frame into the canvas.

This is the scratch buffer.

3. We iterate over scratch buffer, pixel by pixel, passing each pixel to our effects function for processing.



4. After all the pixels in the scratch buffer are processed, we copy them from the scratch buffer canvas to the display canvas.



5. And then we repeat the process on every frame as it is decoded by the video object.

Implementing a scratch buffer with Canvas

As you already know, to implement a scratch buffer in canvas we need two canvases: one to do our computation in, and one to display our results. To create those canvases, we'll start back in our HTML file `videobooth.html`. Open this file and find the `<div>` with the id "`videoDiv`" and add two canvas elements below the `<video>`:

```
<div id="videoDiv">
    <video id="video" width="720" height="480"></video>
    <canvas id="buffer" width="720" height="480"></canvas>
    <canvas id="display" width="720" height="480"></canvas>
</div>
```

We're adding two canvas elements, one for the buffer and one to display.

Notice they're exactly the same size as the video element.

How to position the video and canvases

Now you might be wondering about positioning these elements; we're going to position them right on top of each other. So at the bottom will be the video element, on top of that is the buffer, and on top of that, the display canvas element. We're using CSS to do it, and although we don't talk much about CSS in this book, if you open `videobooth.css` you'll see the positioning for the three elements:

```
div#videoDiv {
    position: relative;
    width: 720px;
    height: 480px;
    top: 180px;
    left: 190px;
}
video {
    background-color: black;
}
div#videoDiv canvas {
    position: absolute;
    top: 0px;
    left: 0px;
}
```

The videoDiv <div> is positioned relative to the element it's in (the console <div>), at 180px from the top and 190px from the left, which places it in the center of the console. We set the width and height equal to the width and height of the <video> and the two <canvas> elements.

The <video> is the first element in the videoDiv <div> so it's automatically positioned at the top left of the <div>. We set the background to black so that if we have letter-boxing or pillar-boxing, the space is black.

The two <canvas> elements in the videoDiv <div> are positioned absolutely with respect to the videoDiv (their parent), so by placing the <canvas> elements at 0px from the top, and 0px from the left, they are in exactly the same position as the <video> and the videoDiv.

Writing the code to process the video

We've got a video element, a buffer that's a canvas, and a canvas that is going to display the final video frames. And we've also got them stacked on one another so we see only the top display canvas, which will contain the video with the effect applied. To process the video we're going to use the video element's `play` event, which is called as soon as a video begins playing. Add this to the end of the `onload` handler:

```
video.addEventListener("play", processFrame, false);
```

When the video begins playing it will call the function processFrame.

The `processFrame` function is where we'll process the video pixels and get them into the canvas for display. We'll start by making sure we have access to all our DOM objects:

```
function processFrame() {
    var video = document.getElementById("video");
    if (video.paused || video.ended) {
        return;
    }
    var bufferCanvas = document.getElementById("buffer");
    var displayCanvas = document.getElementById("display");
    var buffer = bufferCanvas.getContext("2d");
    var display = displayCanvas.getContext("2d");
}
```

First grab the video object...

... and check to see if the video is still playing. If it isn't then we've got no work to do, just return.

Then grab a reference to both canvas elements and also to their contexts, we're going to need those.

How to create the buffer

To create the buffer, we need to take the current video frame, and copy it to the buffer canvas. Once we have it on the canvas, we can process the data in the frame. So, to create that buffer we do this (add this to the bottom of `processFrame`):

It takes an image and draws that image onto the canvas, at an x,y position for a given width and height.

Remember the context drawImage method from Chapter 7?

This time, we're getting an image from the video. By specifying the video as the source, drawImage gets one frame of the video as image data.

```
buffer.drawImage(video, 0, 0, bufferCanvas.width, bufferCanvas.height);
var frame = buffer.getImageData(0, 0, bufferCanvas.width, bufferCanvas.height);
```

Then we grab the image data from the canvas context and store it in a variable, frame, so we can process it.

Here, we're just saying we want all the image data in the canvas.

How to process the buffer

We've got our hands on a frame of video data, so let's do some processing on it! To process the frame, we're going to loop over every single pixel in the frame data and pull out the RGB color values that are stored in each pixel. Actually, each pixel has 4 values, RGB and Alpha (the opacity), but we're not going to use the Alpha. Once we've got the RGB values, we'll call the `effectFunction` (remember, that's the function we set back on page 392 and asked you to tuck in the back of your brain!) with the RGB information and the frame.

Add this code to the bottom of your `processFrame` function:

```
buffer.drawImage(video, 0, 0, bufferCanvas.width, displayCanvas.height);
var frame = buffer.getImageData(0, 0, bufferCanvas.width, displayCanvas.height);

var length = frame.data.length / 4;

for (var i = 0; i < length; i++) {
    var r = frame.data[i * 4 + 0];
    var g = frame.data[i * 4 + 1];
    var b = frame.data[i * 4 + 2];
    if (effectFunction) {
       effectFunction(i, r, g, b, frame.data);
    }
}
display.putImageData(frame, 0, 0);
```

First, we find out the length of the frame data. Notice that the data is in a property of frame, frame.data, and length is a property of frame.data. The length is actually four times longer than the size of the canvas because each pixel has four values: RGBA.

Now we loop over the data and get the RGB values for each pixel. Each pixel takes up four spaces in the array, so we grab r from the first position, g from the second, and b from the third.

Then, we call the effectFunction (if it's not null, which it will be if the "Normal" button is pressed), passing in the position of the pixel, the RGB values, and the frame.data array. The effect function will update the frame.data array with new pixel values, processed according to the filter function assigned to effectFunction.

At this point the frame data has been processed, so we use the context putImageData method to put the data into the display canvas. This method takes the data in frame and writes it into the canvas at the specified x, y position.

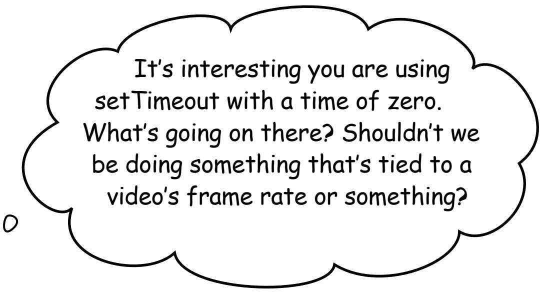
We've processed one frame, what next?

Yes, that's one single frame we just processed, and we want to keep processing them all as the video continues to play. We can use `setTimeout` and pass it a value of zero milliseconds to ask JavaScript to run `processFrame` again as soon as it possibly can. JavaScript won't actually run the function in *zero* milliseconds, but it will give us the next soonest time slot we can get. To do that, just add this to the bottom of your `processFrame` function:

setTimeout is just like setInterval, except that it runs only once after a specified time in milliseconds.

```
setTimeout(processFrame, 0);
```

Tells JavaScript to run processFrame again as soon as possible!



We wish we could.

You're absolutely right: what we'd love to do is have our handler called once for every frame, but the video API doesn't give us a way to do that. It does give us an event named `timeupdate` that can be used to update a running time display of your video, but it doesn't tend to update at a granularity that you can use for processing frames (in other words, it runs at a slower rate than the video).

So instead we use `setTimeout`. When you pass zero to `setTimeout`, you're asking JavaScript to run your timeout handler as soon as it possibly can—and this leads to your handler running as frequently as it possibly can.

But might that be faster than the frame rate? Wouldn't it be better to calculate a timeout close to what is needed for the frame rate? Well, you could, but it's unlikely that the handler is going to actually get to run in lockstep with the frames of your video, so zero is a good approximation. Of course, if you are looking to enhance the performance characteristics of your app, you can always do some profiling and figure out what the optimal values are. But until we have a more specific API, that's our story.

Now we need to write some effects

Finally, we've got everything we need to write the video effects: we're grabbing each frame as it comes in, accessing the frame data pixel by pixel and sending the pixels to our effect filter function. Let's look at the *Film Noir* filter (which, in our version, is just a fancy name for black and white):

The filter function is passed the position of the pixel...

... the red, green, and blue pixel values ...

... and a reference to the frame data array in the canvas.

```
function noir(pos, r, g, b, data) {
    var brightness = (3*r + 4*g + b) >>> 3;
    if (brightness < 0) brightness = 0;
    data[pos * 4 + 0] = brightness;
    data[pos * 4 + 1] = brightness;
    data[pos * 4 + 2] = brightness;
}
```

>>> is a bitwise operator that shifts the bits in the number value over to modify the number. Explore further in a JavaScript reference book.

So the first thing we do is compute a brightness value for this pixel based on all its components (r, b and g).

And then we assign each component in the canvas image to that brightness.

Remember this function is called once per pixel in the video frame!

This has the affect of setting the pixel to a grey scale value that corresponds to the pixel's overall brightness.

A film noir test drive

Add this function to `videobooth.js` and then reload your page. As soon as the video starts rolling press the Film Noir button and you'll see a brooding black & white film look. Now choose Normal again. Not bad, eh? And all in JavaScript, in real time!

Kind of amazing when you think about it.

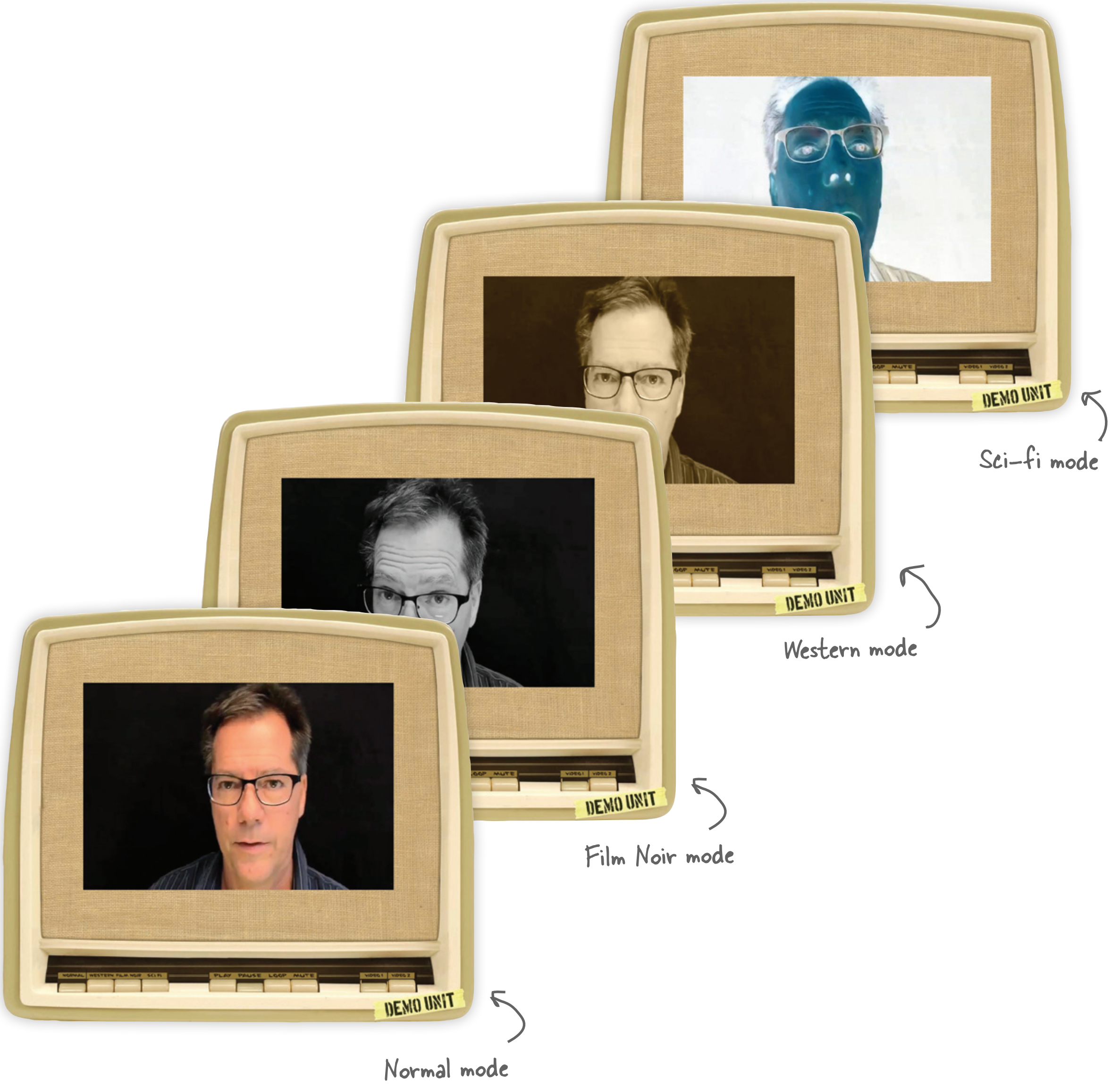
Sharpen your pencil

This book isn't really about video processing and effects, but it sure is fun. Below we've got the western and sci-fi effects. Look through the code and make notes on the right as to how each works. Oh, and we added an extra one—what does it do?

```
function western(pos, r, g, b, data) {
    var brightness = (3*r + 4*g + b) >>> 3;
    data[pos * 4 + 0] = brightness+40;
    data[pos * 4 + 1] = brightness+20;
    data[pos * 4 + 2] = brightness-20;
}

function scifi(pos, r, g, b, data) {
    var offset =  pos * 4;
    data[offset] = Math.round(255 - r) ;
    data[offset+1] = Math.round(255 - g) ;
    data[offset+2] = Math.round(255 - b) ;
}

function bwcartoon(pos, r, g, b, outputData) {
    var offset =  pos * 4;
    if( outputData[offset] < 120 ) {
        outputData[offset] = 80;
        outputData[++offset] = 80;
        outputData[++offset] = 80;
    } else {
        outputData[offset] = 255;
        outputData[++offset] = 255;
        outputData[++offset] = 255;
    }
    outputData[++offset] = 255;
    ++offset;
}
```

The Big Test Drive

This is it! We have this code wrapped up and ready to ship off to **Starring You Video**. Go ahead and double check that you've got all the code typed in, save, and load `videobooth.html`. Then have fun playing around with your new app!

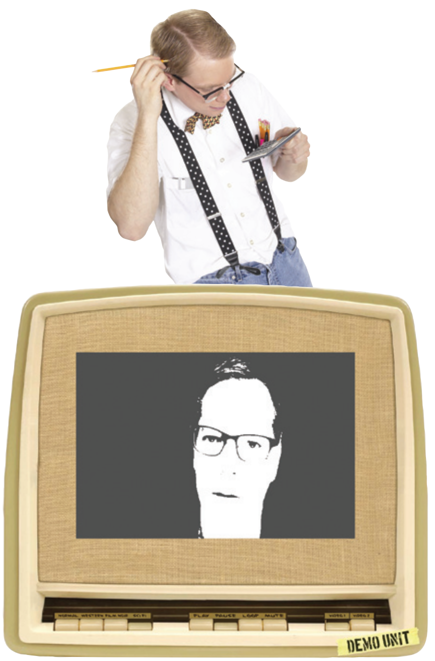


IN THE LABORATORY

Obviously we've only scratched the surface in terms of video processing, and we're sure you can think of more creative effects than those we came up with. Go ahead and think up a few, implement them, and document them here.

Have you invented something really cool and implemented it? Tell us about it at wickedlysmart.com and we'll feature it for other readers!

Your ideas here!



B&W Cartoon is just one of many other fun things you can do with effects.

Hey I know its almost the end of the chapter, but I keep meaning to ask this: we've been loading video from a local file, what changes if my video is hosted on the Web?

Sure, just use a web URL.

You can substitute a web URL for any of the sources we been defining locally. For instance:

```
<video src="http://wickedlysmart.com/myvideo.mp4">
```

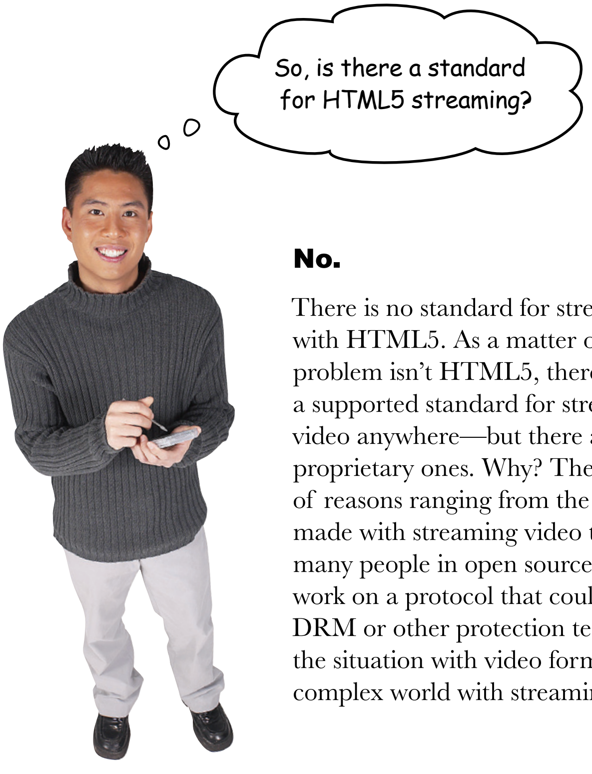
Keep in mind there is more room for bad things to happen when you are delivering on the Web (and we'll talk about how to handle those things in a moment). Also, the bitrate of your videos starts to matter a lot more when delivering to a browser or a mobile device over the network. Like with video formats, if you're going down this road, seek out experts and educate yourself.

Great, and one more question, is there a difference between what we're doing and streaming video?

Yes, a big difference.

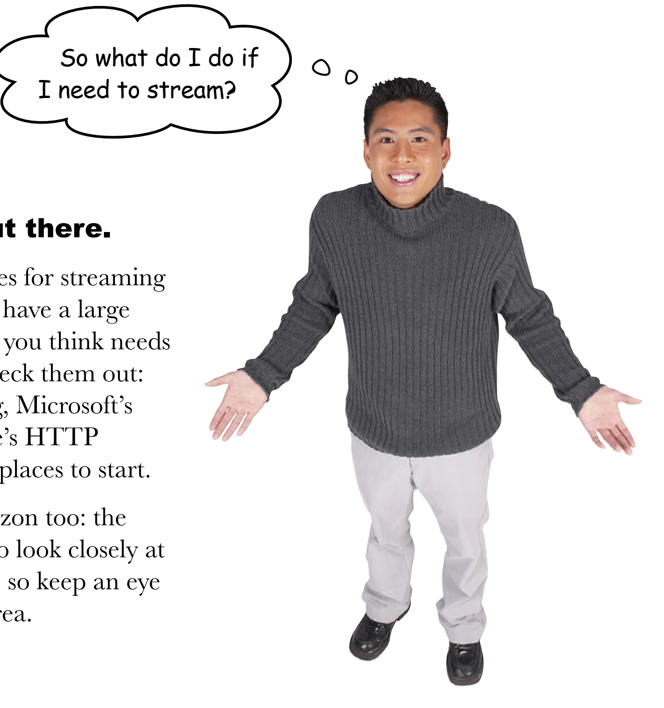
The term streaming often gets used like the term xerox or kleenex—as a generic term for getting video from the Web to your browser. But "progressive video" and "streaming video" are actually technical terms. In this book we've been using progressive video, which means when we retrieve the video (either locally or over the network) we're retrieving a file using HTTP, just like an HTML file or an image, and we try to decode and play it back as we retrieve it. Streaming video is delivered using a protocol that is highly tuned to delivering video in an optimal way (perhaps even altering the bitrate of the video over time as bandwidth becomes more or less available).

Streaming video probably sounds like it would provide your user with a better experience (it does), and is perhaps more efficient in terms of your user's connection and your bandwidth charges (it is). On top of all that, streaming video makes it easier to do things like protect the content of your video if you need that kind of security.



No.

There is no standard for streaming video with HTML5. As a matter of fact the problem isn't HTML5, there isn't really a supported standard for streaming video anywhere—but there are plenty of proprietary ones. Why? There are a number of reasons ranging from the money to be made with streaming video to the fact that many people in open source don't want to work on a protocol that could be used for DRM or other protection technologies. Like the situation with video formats, we're in a complex world with streaming video.



There are solutions out there.

There are lots of legitimate uses for streaming video technologies, and if you have a large audience, or you have content you think needs to be protected, you should check them out: Apple's HTTP Live Streaming, Microsoft's Smooth Streaming and Adobe's HTTP Dynamic Streaming are good places to start.

There's good news on the horizon too: the standards bodies are starting to look closely at HTTP-based video streaming, so keep an eye out for developments in this area.

If only it were a perfect world...

But it's not: we have all those nasty network issues, incompatible devices and operating systems, and an increasing chance of asteroids hitting the earth. That last one we can't help with, but for the first two actually knowing you have an error is half the battle, then you can at least do something about it.

The video object has an error event, which can be thrown for a number of reasons that can be found in the `video.error` property, or more specifically in the `video.error.code` property. Let's take a look at what kinds of errors we can detect:

Errors

MEDIA_ERR_ABORTED=1

Used any time the process of getting the video over the network is aborted by the browser (possibly at a user's request).

MEDIA_ERR_NETWORK=2

Used whenever a network retrieval of the video is interrupted by a network error.

MEDIA_ERR_DECODE=3

Used whenever the decoding of a video fails. This could happen because the encoding uses features the browser can't support or because the file is corrupt.

MEDIA_ERR_SRC_NOT_SUPPORTED=4

Used when the specified video source cannot be supported because of a bad URL or because the source type isn't decodable by the browser.

Each error type also has an associated number that is the error code produced by the error event, we'll see this in just a sec...

How to use error events

Dealing with errors is complex business and how you deal with errors depends a lot on your application, and what would be appropriate for the app and your users. That said, we can at least get you started and point you in the right direction. Let's take Webville TV and give it the ability to know it has encountered an error—and if it does encounter one, give the audience a PLEASE STAND BY message.

We want to be notified when there's an error message, so we need to add a listener for the error event. Here's how we do that (add this to the onload handler in `webville.js`):

```
video.addEventListener("error", errorHandler, false);
```

When an error occurs, the errorHandler function is called.

Now we need to write the function `errorHandler`, which will check if there is an error, and if so, place our "please stand by" image on the video display by making it the poster image:

If the handler is called, we make sure there is an error by checking video.error and then we place a poster up on the video display.

```
function errorHandler() {
    var video = document.getElementById("video");
    if (video.error) {
        video.poster = "images/technicaldifficulties.jpg";
        alert(video.error.code);
    }
}
```

Optionally add this line to be able to see the error code (see the previous page for the integer stored in the code property).

Test Crash!

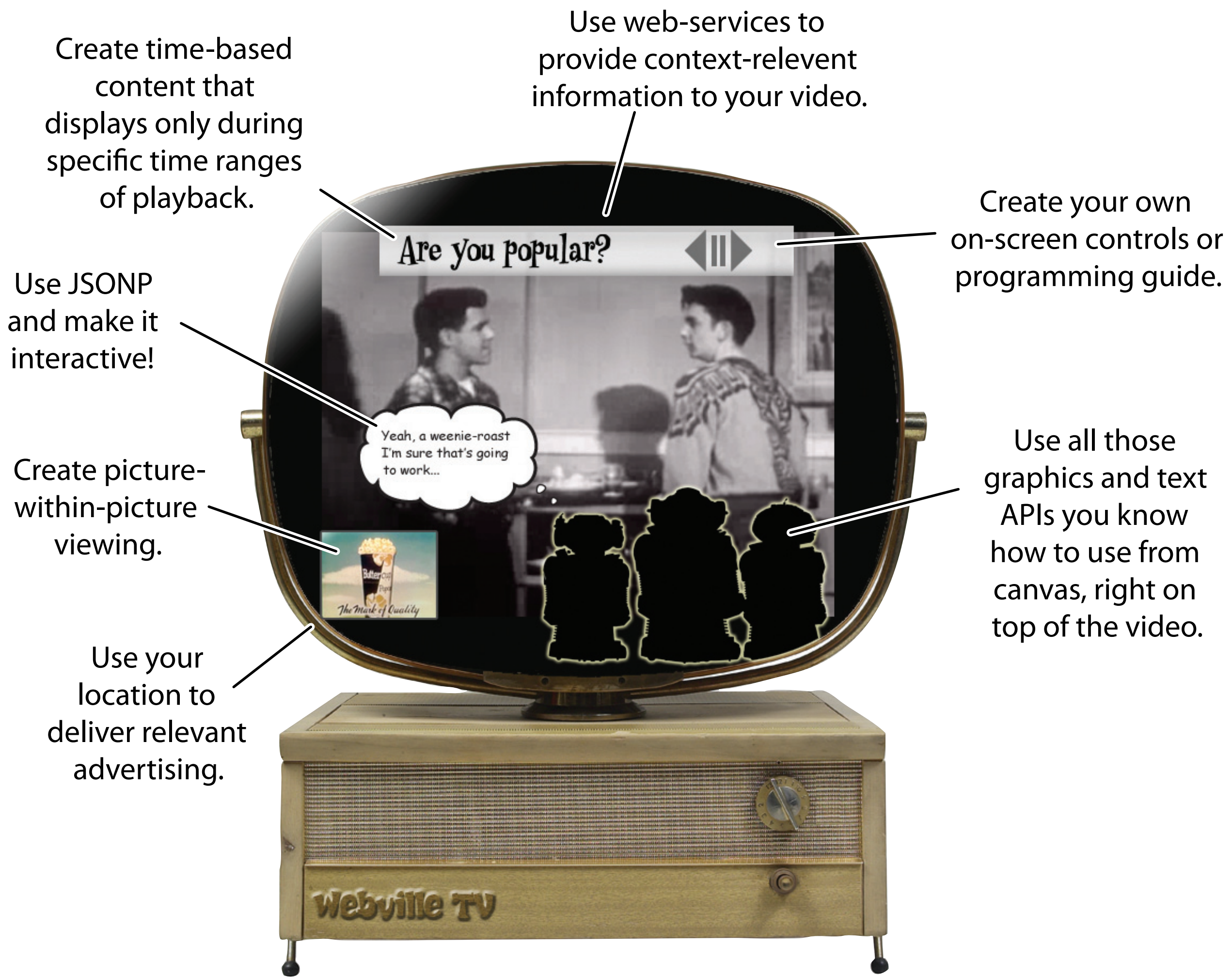
There are many ways for the playback of the video to fail, and to test this code you're going to make it fail. Here are a few suggestions:

- Disconnect your network at different points in playback.
- Give the player a bad URL.
- Give the player a video you know it can't decode.
- Give the player a URL that isn't even a video.
- Use software to reduce your bandwidth (it's out there, just look for it).

So get this code typed in and get testing. Remember you can map the integer in the alert dialog back to a real code by looking at the codes on page 407.

Where can you go from here?

This is where it gets exciting, because think of all you know how to do with HTML markup, with the video element and, of course, the canvas...not to mention web services, geolocation... wow. Sure, we did some cool video processing with canvas, but you can apply everything you know how to do with canvas to video. Here are just a few ideas we had, please add your own. And give yourself a pat on the back from us, you've earned it!

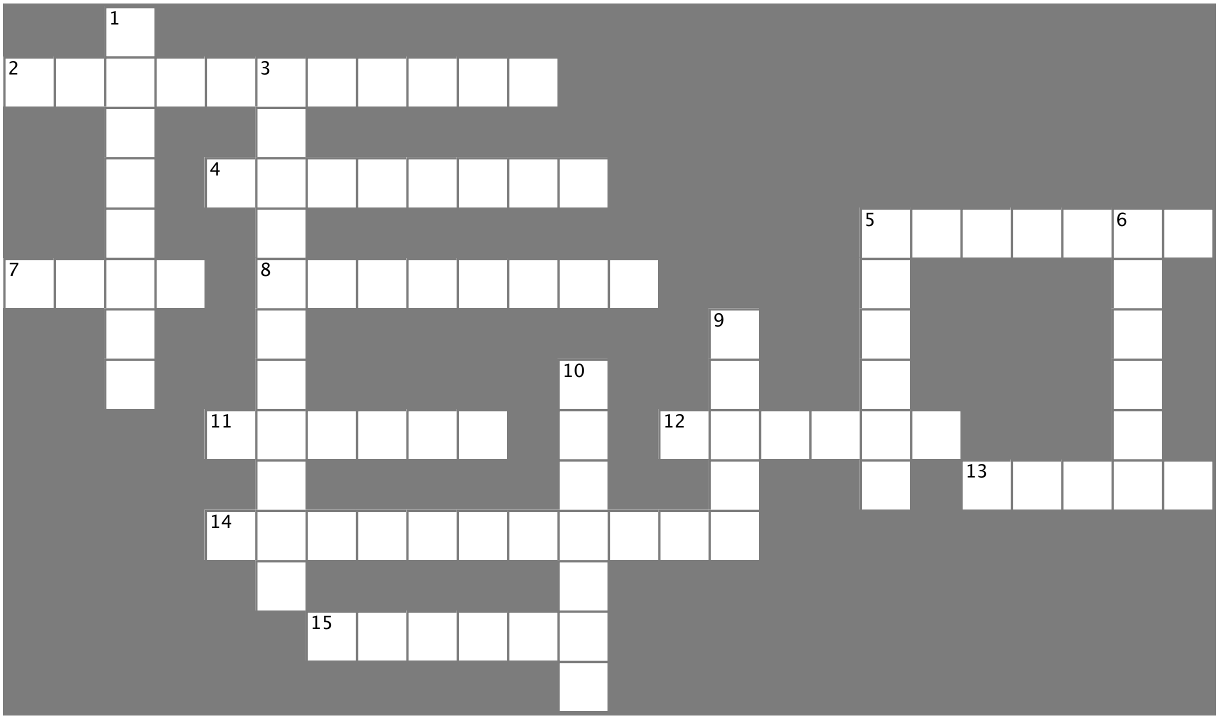


BULLET POINTS

- You can play video by using the <video> element with a few simple attributes.
- The autoplay attribute starts playback on page load, but use only when appropriate.
- The controls attribute causes the browser to expose a set of playback controls.
- The look and feel of controls differs among browsers.
- You can supply your own poster image with the poster attribute.
- The src attribute holds a URL to the video to be played.
- There are many “standards” for video and audio formats.
- Three formats are in common usage, WebM, MP4/H.264 and Ogg/Theora.
- Know your audience to know what formats you need to supply.
- Use the <source> tag to specify alternative video formats.
- Use fully specified types in your <source> tag to save the browser work and time.
- You can continue to support other video frameworks, like Flash, by adding a fallback <object> tag in the video element.
- The video object provides a rich set of properties, methods and events.
- Video supports play, pause, load, loop and mute methods and properties to directly control the playback of the video.
- The ended event can be used to know when video playback has ended (for instance, to implement a playlist).
- You can programmatically ask the video object if it can play a format with canPlayType.
- The canPlayType method returns empty string (no support for format), maybe (if it might be able to play the format) or probably (if it confidently thinks it can play the format).
- Canvas can be used as a display surface for video to implement custom controls or other effects with video.
- You can use a scratch buffer to process video before copying it to the display.
- You can use a setTimeout handler to process video frames; while it isn’t linked directly to every frame of the video, it is the best method we have right now.
- You can use a URL as a video source to play network-based videos.
- Some browsers enforce a same origin policy on video so that you need to serve the video from the same origin as your source page.
- Errors are always possible, if not probable, with video, especpecilly when a network is involved.
- The error event can be used to notify a handler when video retrieval, decoding or playback errors occur.
- The video element relies on progressively downloaded video. Currently there is no HTML5 standard for streaming, although the standards bodies are looking at HTTP-based streaming solutions.
- There is currently no standard way of protecting video delivered through the video element.

HTML5cross

Before you sit back and watch some more Webville TV, do a quick crossword to make it all stick. Here's your Chapter 8 crossword puzzle.



Across

2. Type of delivery the video element uses for video.
4. To provide several video options, use _____ source elements.
5. Kind of buffer we used canvas for.
7. Property to play your video over and over.
8. Starts a video as soon as it can.
11. The open source audio codec.
12. Used to display processed video.
13. When the show is over, this event is thrown.
14. I can play this type, can you?
15. Look and feel of browser controls _____.

Down

1. Use __________ if you want a built-in way to control video.
3. We saw '50s ______ films.
5. What you should do if an asteroid is going to hit the earth.
6. The Starbuzz CEO spills his ___________.
9. What we processed on every setTimeout call.
10. Clint Eastwood would like this effects style.

Sharpen your pencil Solution

This book isn't really about video processing and effects, but it sure is fun. Below we've got the western and sci-fi effects. Look through the code and make notes on the right as to how each works. Oh, and we added an extra one—what does it do? Here's our solution.

```
function western(pos, r, g, b, data) {
    var brightness = (3*r + 4*g + b) >>> 3;
    data[pos * 4 + 0] = brightness+40;
    data[pos * 4 + 1] = brightness+20;
    data[pos * 4 + 2] = brightness-20;
}
```

The Western filter emphasizes the red and green components of the pixel while de-emphasizing the blue component, to give the video a brownish tinge.

```
function scifi(pos, r, g, b, data) {
    var offset =  pos * 4;
    data[offset] = Math.round(255 - r) ;
    data[offset+1] = Math.round(255 - g) ;
    data[offset+2] = Math.round(255 - b) ;
}
```

The scifi filter reverses the amounts of RGB components of each pixel. So if a pixel had a lot of red, it now has a little. If a pixel had a little green, it now has a lot.

```
function bwcartoon(pos, r, g, b, outputData) {
    var offset =  pos * 4;
    if( outputData[offset] < 120 ) {
        outputData[offset] = 80;
        outputData[++offset] = 80;
        outputData[++offset] = 80;
    } else {
        outputData[offset] = 255;
        outputData[++offset] = 255;
        outputData[++offset] = 255;
    }
    outputData[++offset] = 255;
    ++offset;
}
```

The bwcartoon filter turns every pixel with a red component of less than 120 (out of 255) into black, and turns all other pixels into white, giving the video a weird cartoony-like B&W appearance.

VIDEO RECONNAISSANCE SOLUTION

TOP SECRET

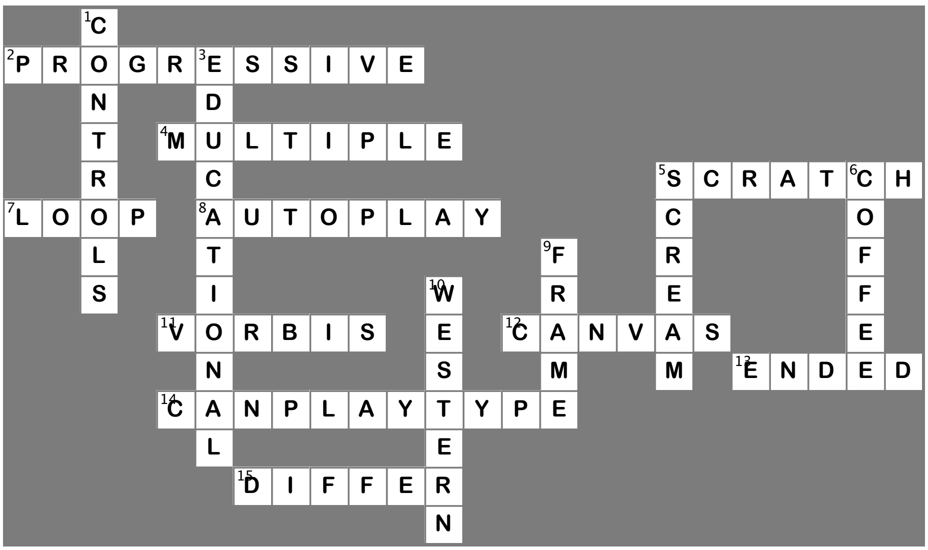
iOS and Android devices (among others)

Video \ Browser	Safari	Chrome	Firefox	Mobile Webkit	Opera	IE9+	IE8	IE7 or <
H.264	✓	some		✓		✓		
WebM		✓	✓		✓			
Ogg Theora		✓	✓		✓			

Watch it!

This will change fast! So check the latest support on the Web.

HTML5cross Solution



9 storing things locally

Web Storage

Tired of stuffing your client data into that tiny ~~closet~~ cookie? That was fun in the '90s, but we've got much bigger needs today with web apps. What if we said we could get you five megabytes on every user's browser? You'd probably look at us like we were trying to sell you a bridge in Brooklyn. Well, there's no need to be skeptical—the HTML5 Web Storage API does just that! In this chapter we're going to take you through everything you need to store any object locally on your user's device and to make use of it in your web experience.

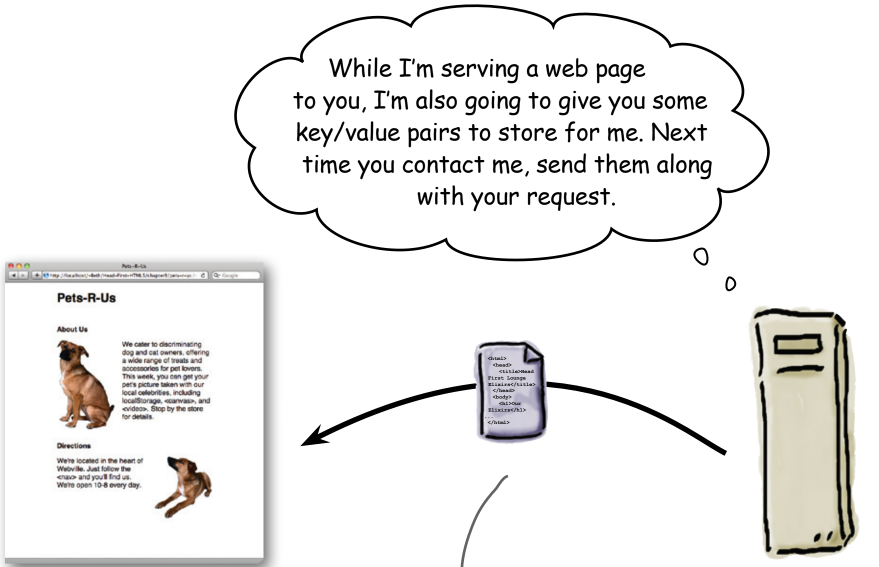
How browser storage works (1995 - 2010)

Behind the Scenes

Building a shopping cart? Need to store some user preferences for your site? Or just need to stash some data that you need to be associated with each user? That's where browser storage comes in. Browser storage gives us a way to persistently store data that we can use in building a web experience.

Up until now there's been one game in town—the browser cookie—for storing information on the browser. Let's see how cookies work:

1. When your browser retrieves a web page, say from "pets-R-us.com," the server can send a cookie along with its response. Cookies contain one or more key and value pairs:



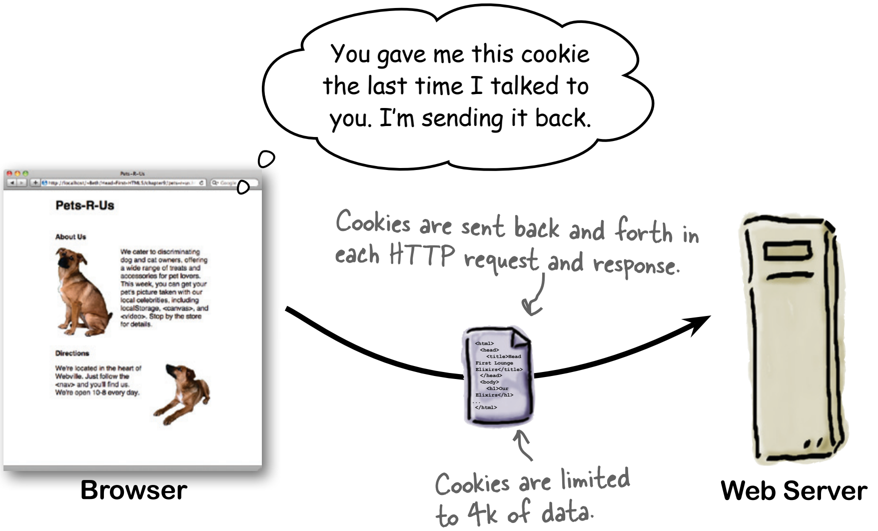
The browser saves the cookie locally and will send it back to the server the next time it makes a request.

The cookie.

```
Cookie: pet=dog; age=5; color=black
```

Here are some key and value pairs. We've got a key of "pet" with a value of "dog", and a key of "age" with a "5", and so on...

2 The next time the browser makes a request to "pets-R-us.com," it sends along any cookies that were sent previously:



---

3 The server can then use the cookie to personalize the experience, in this case promoting relevant items to the user, but there are many other ways cookies can be used too.

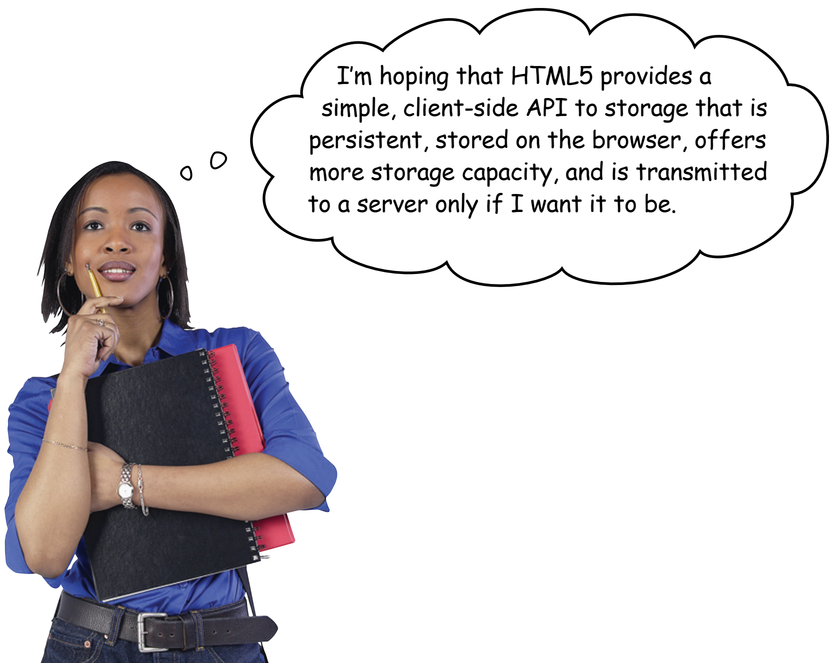
BRAIN POWER

Cookies have been with us a long time, but you might be able to think of some ways they could be improved on.

Check all the items below that you think make cookies problematic:

- [ ] There's only 4k to work with, my app needs more storage than that.
- [ ] Sending the cookie back and forth every time seems really inefficient, especially if I'm on a mobile device with not a lot of bandwidth.
- [ ] They sound like a good way to transmit viruses and other malware to my browser.
- [ ] I've heard the way the key/value pairs are done as part of the HTTP request is a pain to deal with in code.
- [ ] Aren't we potentially sending personal data back and forth every time we make a request?
- [ ] They don't seem well matched to all the client-side development we've been doing. They seem to assume everything in happening in the server.

For the record, and despite news reports to the contrary, cookies are quite safe and not a haven for virus writers.

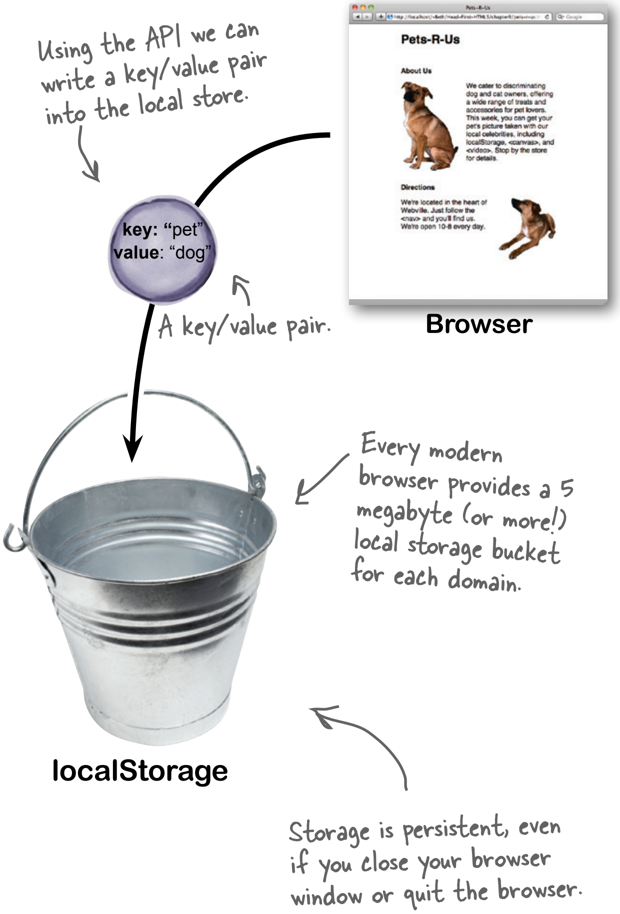


How HTML5 Web Storage works

Behind the Scenes

HTML5 gives us a nice, simple JavaScript API in the browser for storing key/value pairs that are persistent. You're not limited to four stingy kilobytes of storage either; all browsers today will gladly offer you five to ten megabytes of storage in every user's browser. HTML5's local storage was also created with web apps (and mobile apps!) in mind—local storage means your app can store data in the browser to reduce the communication needed with the server. Let's check out how it works (and then we'll jump head first into the API):

1. A page can store one or more key/value pairs in the browser's local storage.



2. And then later use a key to retrieve its corresponding value.

NOTE: The server still serves your pages, and you may even send a bit of the data in your local store to the server for some server-side computation, but the client is dealing with the details of local storage, not the server (as is common with cookies).

Note to self...

Need a system for getting things done? It's hard to improve on the old Post-it note system (more commonly known as *stickies*). You know how it works: you jot down your "to do" item, stick it somewhere, and once you've done the task, you throw the sticky in the trash (or recycle it).

High-tech productivity tool.

How about we build one using HTML? Let's see, we need a way to store all those stickies, so we're going to need a server, and some cookies... oh, wait a second, back up the bus, we can do this with the HTML5 Web Storage API!

The Web Storage API is simple, fun and instantly gratifying. We promise!

No fooling around, we're going to jump right in and start using the local store. To do that you should create a simple html page with all the basics: a head, a body, and a script (or just use the starter file `notetoself.html` in the code examples). Follow along by typing the code into your <script> element (typing it in helps it stick):

1. There's not much more to a sticky than the text you write on it, right? So, let's start by storing a sticky for "Pick up dry cleaning":

We're starting simple, but before you know it, we'll have a whole Stickies app up and running.

The Web Storage API is available to you through the localStorage object. You'll find this already defined for you by the browser. When you use it you're making use of the underlying local storage system.

The setItem method takes two strings as arguments that act as the key/value pair.

You can only store items of type String. You can't directly store numbers or objects (but we'll find a way to overcome this limitation soon).

```
localStorage.setItem("sticky_0", "Pick up dry cleaning");
```

To store something, we use the setItem method.

The first string argument is a key that the item is stored under. Name it whatever you want as long as it is a string.

The second string is the value you'd like to store in local storage.

2. That was easy enough; let's add a second item to the local store:

```
localStorage.setItem("sticky_1", "Cancel cable tv, who needs it now?");
```

Another key. Like we said already, you can use any key you like as long as it is a string, but you can only store one value per key.

A value to go with our new key.

3. Now that we have two values stored safely in our browser's local storage, you can now use one of the keys to retrieve its corresponding value from localStorage. Like this:

We're getting the value associated with the key "sticky_0" from the local store...

...and assigning it to the variable named sticky.

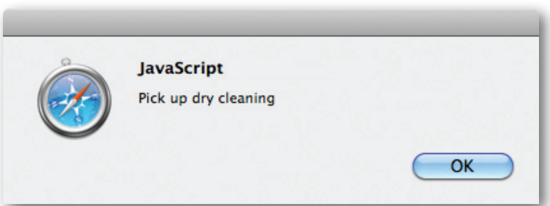
```
var sticky = localStorage.getItem("sticky_0");

alert(sticky);
```

And to make this a little more interesting, let's use the alert function to pop the sticky note's value up on the screen.

Time for a test drive!

Make sure you've got all this code into your script element and load it into your browser.

Here's the result of our test drive:



There's our JavaScript alert, with the value of sticky_0 as the alert message.

What's cool about this is that this value was stored in and retrieved from the browser's localStorage! You could quit your browser, go on vacation to Fiji for a month, come back, and it will still be there waiting on you.

Okay, okay, we agree the example could have been a little more exciting, but work with us here, we're getting there...

Sure. Here it is in a nutshell: your browser provides you with a local store—space on your own computer, in your browser—that a page can make use of to store key/value pairs. You created a few key/value pairs, stored them away using the local storage API, and then you retrieved one of them for use in your app. Now, while that might not be the most exciting example, there are lots of interesting things you can do with a bit of storage in every user's browser (and we're sure you can think of at least a few).

So now that you've got an answer in a nutshell, let's step through in detail what just happened:

1. First, remember every browser has a bit of local storage that you can use to store key/value pairs.

Every modern browser has local storage behind the scenes ready for you to use to store key/value pairs.

2 With that local storage you can take a key and a value (both in the form of strings) and you can store them.

```
localStorage.setItem("sticky_0", "Pick up dry cleaning");
```

We use the setItem method to store a key/value pair. The key is “sticky_0” and the value is “Pick up dry cleaning”.

Key/value pair created by calling setItem.

Once you’ve placed the key/value pair in localStorage, it is persistently stored for you, even if you close the browser window, quit your browser or reboot your computer.

localStorage

3 We then called setItem again and stored a second key/value pair, this time with a key of “sticky_1” and a value of “Cancel cable tv, who needs it now?”.

```
localStorage.setItem("sticky_1", "Cancel cable tv, who needs it now?");
```

Now there are two values stored under two unique keys.

localStorage

4 And when we call getItem with a key of “sticky_0”, it returns the value of the key/value pair.

```
localStorage.getItem("sticky_0");
```

returns

“Pick up dry cleaning”

getItem finds an item with a key equal to “sticky_0” (if it exists) and returns its value.

Note that by getting an item we don’t remove it from the store, it’s still there. We’re just getting the value for the given key.

there are no Dumb Questions

Q: First you said "Web Storage" and then you started talking about "local storage." Are they the same?

A: The Web standard is named "Web Storage" but most people just call it local storage (in fact, the browsers even expose the API through the localStorage object). Web Storage is actually not the best name for the standard (because items are stored in your browser, rather than on the Web). But that said, we're stuck with it. You'll see us use the term local storage more than the standard name of "Web Storage."

Q: How widely supported is the Web Storage API? Can I count on it being there?

A: Yes; in fact it is one of the better supported APIs, even all the way back to IE8 and is now in most modern mobile browsers. There are a few caveats here and there, but we'll point them out as we go. In terms of counting on Web Storage, as always you should test before using APIs. Here's how you test for localStorage:

```
if (window["localStorage"]) {
   // your localStorage code here...
}
```

Notice that we test by checking to see if the window global object has the localStorage property. If it's there, we know the browser supports localStorage.

Q: At the very beginning of the chapter you mentioned 5MB of storage on each browser. Is that five megabytes total across all apps?

A: No, that is actually five megabytes per domain.

Q: You said the server didn't need to be involved, but then you started talking about domains.

A: Right, all the storage is managed in the client. The domain comes in because five megabytes is allocated to all the pages from the same domain for storage. Pet-R-Us.com gets five, PetEmporium.com gets five more, and so on, all on your machines.

Q: How does this compare to Google Gears [or insert your favorite proprietary local storage technology here]?

A: There's nothing wrong with other browser storage technology, but HTML5's local storage is now the standard (and Google, Apple, Microsoft and others now recognize Web Storage as the standard way to store content locally in the browser).

Q: What happens if I perform a `setItem` on the same key multiple times. Say I called setItem twice on `"sticky_1"`, what happens? Do I get two `sticky_1`'s in the local store?

A: No. Keys are unique in localStorage, so setItem will overwrite the first value with the second value. Here's an example; if you ran this code:

```
localStorage.setItem("sticky_1", "Get Milk");
localStorage.setItem("sticky_1", "Get Almond Milk");
var sticky = localStorage.getItem("sticky_1");
```

The value of sticky would be "Get Almond Milk".

Q: Who can see the data in my local store?

A: Local storage is managed according to the origin (you can just think of the origin as your domain) of the data. So, for instance, every page on wickedlysmart.com can see the items stored by other pages on that site, but code from other sites, say, google.com, can't access that storage (they can access only their own local storage items).

Q: When I'm loading a page from my computer, like we are in these exercises, what is my origin?

A: Good question. In that case your origin is known as the "Local Files" origin, which is great to use for testing. If you have access to server you could test your files there too, and then you'll be in your domain's origin.

Watch it!

Local Storage may not work properly in all browsers if you're using file://.

This is another case where some browsers require that you serve pages using localhost:// or a hosted server, rather than loading from a file. So if your stickies aren't working, try running from a server or try a different browser.

So, I can store strings in localStorage, but what if I want to store a number? I was thinking I might use localStorage to store integer item counts and floating point prices for a shopping cart app I want to write, is this the wrong technology?

Joel

You've got the right technology.

It's true, with `localStorage` you can only use strings as keys and values. But, that's not as restricting as it sounds. Let's say you need to store the integer 5. You can store the string "5" instead, and then convert it back to an integer when you retrieve it from the local store. Let's take a look at how you'd do this for integers and floats.

Say you want to store an integer with the key "numitems". You'd write:

```
localStorage.setItem("numitems", 1);
```

What? Didn't we just say we couldn't store integers?

Okay, it might look like you're storing an integer here, but JavaScript knows this needs to be a string, so it coerces the integer value into a string for you. What `setItem` actually sees is the string "1", not an integer. JavaScript isn't as smart when you retrieve a value with `getItem`:

```
var numItems = localStorage.getItem("numitems");
```

In this code, `numItems` is assigned the string "1", not an integer as we'd like. To make sure `numItems` is a number, you need to use the JavaScript function `parseInt` to convert a string to an integer:

We wrap the value in a parseInt call, which converts the string to an integer.

```
var numItems = parseInt(localStorage.getItem("numitems"));
numItems = numItems + 1;
localStorage.setItem("numitems", numItems);
```

We can add 1 to it because it's a number.

Then we store it again, with JavaScript taking care of the conversion again.

If you're storing floating point values, you'll want to use the `parseFloat` function when you get the price items from localStorage instead:

Same thing here, we store a float value which is coerced into a string.

```
localStorage.setItem("price", 9.99);
var price = parseFloat(localStorage.getItem("price"));
```

And we convert it back to a float with parseFloat.

Were Local Storage and the Array separated at birth?

Local storage has another side you haven't seen yet. Not only does `localStorage` provide the getter and setter methods (that is, `getItem` and `setItem`), it also allows you to treat the localStorage object as an associative array. What does that mean? Well instead of using the `setItem` method, you can assign a key to a value in the store like this:


```
localStorage["sticky_0"] = "Pick up dry cleaning";
```

Here, the key looks like an index for the storage array.

And here's our value sitting over here on the righthand side of an assignment statement.

We can also retrieve the value stored in a key this way too. Here's the syntax:

```
var sticky = localStorage["sticky_0"];
```

This works exactly like using the call to the getItem method.

Here we assign our variable sticky to...

...the value of the key "sticky_0" in the local store.

Not bad, huh? So, use either syntax, they are both valid. But if you are used to using associative arrays in JavaScript, this syntax may be more concise and readable for you.

But wait, there's more!

The localStorage API also provides two other interesting things: a property, `length`, and a method, `key`. The `length` property holds the number of items in the local store. You'll see what the `key` method does below:

Here we're iterating over each item.

The length property tells us how many items are in localStorage.

Big picture: we're using the length to iterate over the contents of localStorage (just like an array), and accessing each key (like "sticky_0") as we go. We can then use that key to extract its corresponding value.

```
for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage[key];
        alert(value);
}
```

For each item in the localStorage, the key method gives us the key (like "sticky_0", "sticky_1" and so on).

Go ahead give it a try...do you get an alert for each item?

Then with the key name we can retrieve the value.

there are no Dumb Questions

Q: When I iterate through localStorage using localStorage.length and localStorage.key, what order are the items in? The same as the order I wrote them into the store?

A: Actually the order of the items isn't defined. What does that mean? It means you'll see every key/value in the store by iterating, but you shouldn't count on any specific order in your code. In fact, different browers may give you different ordering for the same code and items.

The Shell Game

Ready to try your luck? Or should we say skill? We've got a game for you to test your command of localStorage, but you'll need to be on your toes. Use your knowledge of getting and setting key/value pairs in localStorage to keep track of the pea as it shifts from shell to shell.

Feel free to use this space to keep track of the state of localStorage.

```
function shellGame() {
    localStorage.setItem("shell1", "pea");
    localStorage.setItem("shell2", "empty");
    localStorage.setItem("shell3", "empty");
    localStorage["shell1"] = "empty";
    localStorage["shell2"] = "pea";
    localStorage["shell3"] = "empty";
    var value = localStorage.getItem("shell2");
    localStorage.setItem("shell1", value);
    value = localStorage.getItem("shell3");
    localStorage["shell2"] = value;
    var key = "shell2";
    localStorage[key] = "pea";
    key = "shell1";
    localStorage[key] = "empty";
    key = "shell3";
    localStorage[key] = "empty";

    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        alert(key + ": " + value);
    }
}
```

You can type it in to check your answer and see which shell the pea is in.

Which shell has the pea? Write your answer here:

Key	Value
shell1	
shell2	
shell3	

Tonight's talk: **Cookie and Local Storage**

Tonight we have the incumbent browser storage technology, the "Cookie" along with the new front runner, Local Storage.

Cookie: There he is, the golden boy, Local Storage. I've been in this business for over a decade and you think you can come along like you know something. A little wet behind the ears, aren't you?

Local Storage: Sure, you could look at it that way, or, you could say I was built from all the experience gained from your mistakes.

Cookie: Do you have any idea how many pages I'm used on? Ever looked at your stats?

Local Storage: Give it a few years and take another look. The reality is I'm helping to enable a whole new generation of web applications in the browser. A lot of those pages you mention, are *just* pages.

Cookie: Hey, I'm ubiquitous, pervasive, everywhere! I don't think there is a browser on a desktop, device or mobile browser no matter how old, where you won't find me.

Local Storage: I'm catching up fast. Of all the HTML5 technologies, I'm one of the best supported.

Cookie: We'll see. Just what exactly do you think you offer over me? My storage works just fine.

Local Storage: Well, I'm not sure I want to mention this in public, but you do have a size issue.

Cookie: I have no idea what you're talking about.

Local Storage: Hey, you started all this, not me. You know very well that you are limited to 4K of storage, I have over 1,200 times that!

Cookie:	**Local Storage:**
Yeah, I'm light, nimble, we might even say agile.	
	Ha, that's rich. Have you ever talked to a web developer? You're anything but agile. Given you are Mr. Statistics, do you have the stats on the number of developer hours lost to stupid mistakes and misconceptions using cookies?
Come on, I'm an open book, just pure storage to put whatever you want in.	
	What you really mean is you essentially have no data format at all, so developers have to reinvent a new scheme for storing data in cookies.
Oh, and key/value pairs are some great innovation?	
	We don't need great innovation on storage; key/value pairs work great, are straightforward and fit many computing applications.
<Snicker> Oh yeah, and you store everything as a string! Nice work! </Snicker>	
	You can get a lot of mileage out of strings, and if you need something more complex there are ways.
Yeah yeah, call me in ten years, we'll see if you've stood the test of time.	
	Oh you can bet on it. Face it, you were doomed from the start. I mean come on, who names their kid Cookie?
You'll see, you'll be calling me crying when they say "Haha, 5 megabytes, is that all you got?"	

Getting serious about stickies

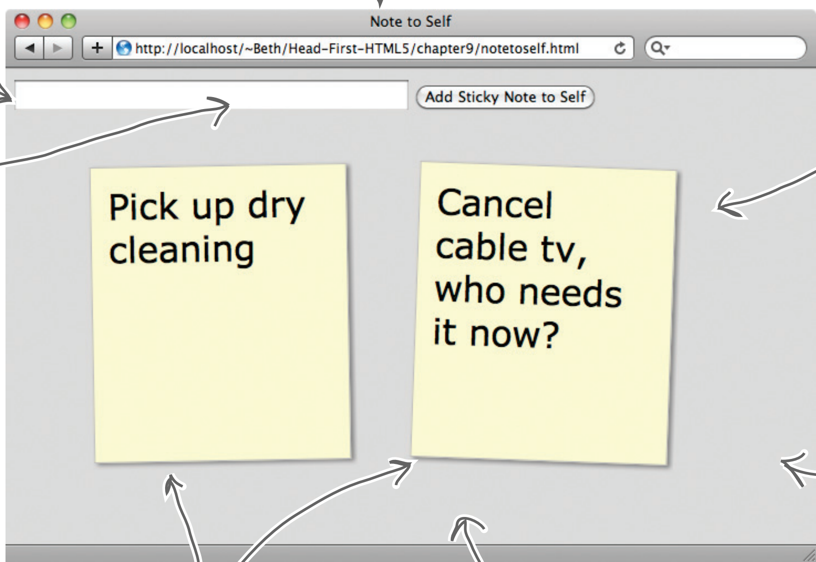
Now that you've had a little time to play with Web Storage, let's take this implementation further. We're going to create a Sticky Notes application so you can see your stickies and add new ones. Let's take a peek at what we're going to build before we build it.

localStorage

The Sticky Notes app will show us notes in localStorage and let us add new ones.

We need a way to add new stickies. So we'll create a form with an input and a button.

If we have existing stickies in storage, we want to see them when we load the page. Like the two stickies we already have.



When you click Add Sticky Note to Self, a new sticky will be added to localStorage.

We'll also see the display updated with any new stickies, and we'll accomplish that by adding a new element to the DOM for each sticky.

To display stickies, we'll iterate through all the stickies in localStorage, and add them to the DOM.

Remember, the keys for these two stickies are "sticky_0" and "sticky_1". We're going to keep following our convention and create keys for stickies with incrementing integers, like sticky_2, sticky_3, and so on.

We'll style the stickies using CSS so they look like real sticky notes!

Creating the interface

To start, we need a way to enter the text of our sticky notes. And it would be great if we could see them in the page, so we need an element to hold all the notes in the page.

Let's work on some code to do that, starting with the HTML markup—take your existing HTML file and add a `<form>` element, the `<ul>` element and the CSS link to it, like below:

Here's our main HTML file.

```
<!doctype html>
<html>
<head>
<title>Note to Self</title>
<meta charset="utf-8">
<link rel="stylesheet" href="notetoself.css">
<script src="notetoself.js"></script>
</head>
<body>
    <form>
        <input type="text" id="note_text">
        <input type="button" id="add_button" value="Add Sticky Note to Self">
    </form>

    <ul id="stickies">
    </ul>
</body>
</html>
```

We've thrown in a little CSS to make things look a little more like real stickies. This book isn't about CSS, but feel free to check out the source!

We're going to move all our JavaScript to the file "notetoself.js".

We've added a form as a user interface to enter new stickies.

And we've got to have somewhere to place our stickies in the interface, so we're going to put them in a unordered list.

The CSS handles making each list item look a little more like a Post-it note.

Now let's add the JavaScript

We've got everything we need in the page now, and we've got a couple sticky notes in localStorage waiting to be displayed. Let's get them on the page by first reading them from localStorage and then placing them inside the unordered list element we just created. Here's how we do that:

When the page is loaded we're going to call the init function...

...which reads all the existing stickies from localStorage and adds them to the <ul> through the DOM.

To do that we iterate over all items in the store.

```
window.onload = init;

function init() {
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        if (key.substring(0, 6) == "sticky") {
            var value = localStorage.getItem(key);
            addStickyToDOM(value);
        }
    }
}
```

Grab each key.

And then we make sure this item is a sticky by testing to see if its key begins with "sticky". Why do we do that? Well, there might be other items stored in localStorage other than our stickies (more on this in a bit).

If it's a sticky, then grab its value and add it to our page (via the DOM).

So now we need to write the `addStickyToDOM` function, which is going to insert the notes into the `<ul>` element:

We're being passed the text of the sticky note. We need to create a list item for the unordered list and then insert it.

```
function addStickyToDOM(value) {
    var stickies = document.getElementById("stickies");
    var sticky = document.createElement("li");
    var span = document.createElement("span");
    span.setAttribute("class", "sticky");
    span.innerHTML = value;
    sticky.appendChild(span);
    stickies.appendChild(sticky);
}
```

So, let's get the "stickies" list element.

Create a list element, and give it a class name of "sticky" (so we can style it).

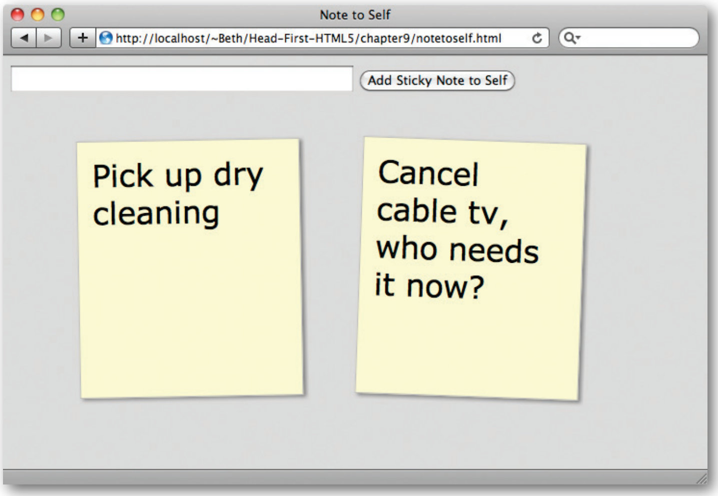
Set the content of the span holding the text of the sticky note.

And add the span to the "sticky" li, and the li to the "stickies" list.

Time for another test drive!

Go ahead and get this code into your script element and load it into your browser.

Here's what we got when we loaded the page in our browser:



Completing the user interface

Now all we need to do is enable the form so we have a way to add new notes. To do that we need to add a handler for when the "Add Sticky Note to Self" button is clicked, and also write some code to create a new sticky. Here's our code to add a handler:

Add this new code to your init function:

```
function init() {
    var button = document.getElementById("add_button");
    button.onclick = createSticky;

    // for loop goes here
}
```

Let's grab a reference to the "Add Sticky Note to Self" button.

And add a handler for when it is clicked. Let's call the handler createSticky.

The rest of the code in init stays the same, we're saving a few trees by not repeating it here.

And the code to create a new sticky note:

When the button is clicked, this handler is invoked.

It first retrieves the text in the form text box.

```
function createSticky() {
    var value = document.getElementById("note_text").value;
    var key = "sticky_" + localStorage.length;
    localStorage.setItem(key, value);

    addStickyToDOM(value);
}
```

Then we need to create a unique key for the sticky. Let's use "sticky_" concatenated with the length of the entire store; it will keep increasing, right?

Then we add a new sticky to localStorage using our key.

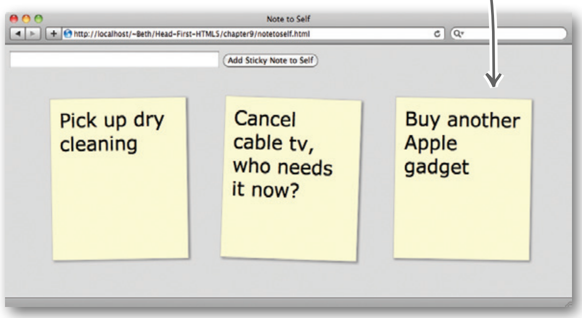
And finally, we add the new text to the DOM to represent the sticky.

Yet another test drive! 

Now we're truly interactive! Load this new code in your browser, enter a new "sticky note to self" and click or tap the "Add Sticky Note to Self" button. You should see the new sticky note appear in your list of stickies. Here's what we see:

You can take that trip to Fiji now, and when you come back, your stickies will still be there waiting for you!

The key for this sticky is "sticky_2", the length of the store (before we added it) concatenated with "sticky_".



Here's our test run! Looks good!

Make sure you try closing your browser window and then opening the file again. Still see the stickies?

there are no Dumb Questions

Q: Why do we test to see if each item's key begins with the string "sticky"?

A: Remember that all the pages from one domain (like apple.com) can see every item stored from other pages in that domain. That means if we aren't careful about naming our keys, we could clash with another page that is using the same keys in a different way. So, this is our way of checking to make sure an item is a sticky (as opposed to say an order number or a game level) before we use its value for a sticky note to self.

Q: What if there are lots of items in localStorage, including lots of items that aren't stickies? Wouldn't it be inefficient to iterate through the entire set of items?

A: Well, unless you are talking about a very large number of items we doubt you'd notice a difference. That said, you're right, it isn't efficient and there may be better ways to approach managing our keys (we'll talk about some of them shortly).

Q: I'm wondering about using localStorage.length as the sticky number in the key. As in

```
"sticky_" + localStorage.length
```

Why did we do that?

A: We need some way to create new keys that are unique. We could use something like the time or generate an integer that we increase each time. Or, as we did, we can use the length of the store (which increases each time we add an item). If you are thinking this might be problematic, we'll come back to that. And if you hadn't thought about it being problematic, no worries, we'll still come back to it.

Q: I created a bunch of stickies in Safari and then switched to Chrome, and I don't see any of my stickies in Chrome. Why not?

A: Each browser maintains its own local storage. So if you create stickies in Safari, you will only see them in Safari.

Q: I just reloaded my page and now my stickies are in a different order!

A: When you add a new sticky note, we add the new sticky note item by appending it to the notes list, so it always goes at the end of the list. When you reload the page, the notes are added in the order they're found in localStorage (which, remember, isn't guaranteed to be in any particular order). You might think that the order would be the same order that the items were added to the store, or some other reasonable ordering, however, you can't count on that. Why? Well one reason is the spec doesn't specify an ordering, so different browsers may implement this in different ways. If your browser does appear to return items in an order that makes sense to you, consider yourself lucky, but don't count on that ordering because your user's browser may order your items another way.

Q: I often use the "for in" form of the for loop. Will that work here?

A: Sure will. It looks like this:

This will iterate through each key in localStorage. Very handy.

```
for (var key in localStorage) {
    var value = localStorage[key];
}
```

Q: What if I don't want a sticky any more? Can I delete stickies?

A: Yes, we can delete items from localStorage using localStorage.removeItem method. You can also remove items from localStorage directly using the browser console. We're going to show you both in this chapter.

BRAIN POWER

Given the way stickies are implemented, there would be a problem with our naming scheme if a user could delete a sticky at will. Can you think of what the problem is?

We need to stop for a little scheduled service

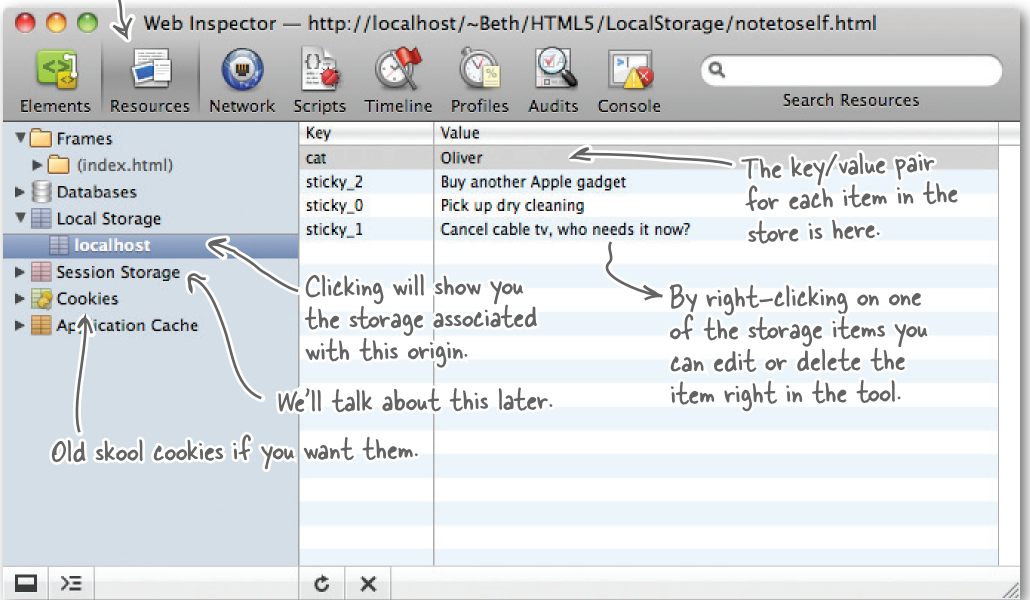
Wouldn't it be great if there were a tool to directly view the items in your localStorage? Or a tool to delete items or even clear the whole thing out and start over when you are debugging?

Well, all the major browsers ship with built-in developer tools that allow you to directly examine your local store. As you might expect, these tools differ between browsers, so rather than covering them all here, we're going to point you in the right direction, and then you can dig in and figure out the specifics of your own browser. As an example though, let's see what Safari offers:

Not to mention new versions of the browsers are popping up faster than we can write pages!

We've clicked on the Resources tab to inspect localStorage.

Developer tools as they appear in the Safari browser.



The key/value pair for each item in the store is here.

Clicking will show you the storage associated with this origin.

By right-clicking on one of the storage items you can edit or delete the item right in the tool.

We'll talk about this later.

Old skool cookies if you want them.

The origin of the storage. Here we're using local files served from http://localhost, but this might also be a domain name if you are testing on a hosted server.

In Safari, we can use these tools to reload the Storage view, and delete a selected item.

To enable or access the developer tools, as we said, you'll need to do different things for different browsers. Point your browser to `http://wickedlysmart.com/hfhtml5/devtools.html` to see how to do this on your specific browser.

Do-It-Yourself maintenance

There's another way to clear out your items (and as we we'll see in a bit, to delete them one by one), which requires doing a little maintenance on your own, right from JavaScript. The localStorage API includes a handy method, `clear`, that deletes all items from your local store (at least, the ones from your domain). Let's take a look at how we can use this call in JavaScript by creating a new file named `maintenance.html`. Once you've done that, add the code below, and we'll step through how it works.

This is a good tool for your toolbox.

```
<!doctype html>
<html>
<head>
<title>Maintenance</title>
<meta charset="utf-8">
<script>
window.onload = function() {
    var clearButton = document.getElementById("clear_button");
    clearButton.onclick = clearStorage;
}

function clearStorage() {
    localStorage.clear();
}
</script>
</head>
<body>
    <form>
        <input type="button" id="clear_button" value="Clear storage" />
    </form>
</body>
</html>
```

We've added one button to the page, and this code adds a click handler for the button.

When you click the button, the clearStorage function is called.

All this function does is call the localStorage.clear method. Use with caution as it will delete <u>all</u> the items associated with the origin of this maintenance page!

And here's our button. Use this file whenever you need to erase everything in localStorage (good for testing).

After you've typed in the code, go ahead and load it in your browser. It's safe (with regards to our Sticky Notes app) to go ahead and clear your localStorage now, so give it a try! Make sure you've figured out your developer tools first so you can observe the changes.

Watch it!

This deletes all items in your domain!

If you've got a super valuable local store related to another project in the same domain, you'll lose all your items by running this code. Just sayin'...

I've got an issue. While I've been doing the exercises in the book, I've also been using my knowledge to create our company's new shopping cart. My Sticky Notes app stopped working. When I look at localStorage with the Safari dev tools, I see that my sticky counts are all messed up, I have "sticky_0", "sticky_1", "sticky_4", "sticky_8", "sticky_15", "sticky_16", "sticky_23", "sticky_42".

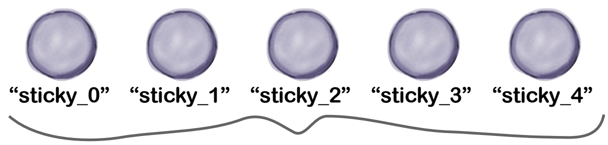
I have a feeling this is happening because I'm creating other items in localStorage at the same time as the stickies. What the heck is going on?!

Ah, you've discovered a major design flaw.

Alright, it's time to come clean: we've built a great little app so far, and it should work perfectly for years to come *as long as you don't introduce any other items* into the localStorage (like Joel did with his shopping cart). Once you do that, our whole scheme of tracking stickies no longer works, or, at least, no longer works well. Here's why:

If you're willing to live with that, cool; otherwise you better keep reading.

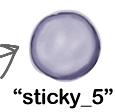
First of all, our sticky notes are numbered from zero to the number of stickies (minus one):



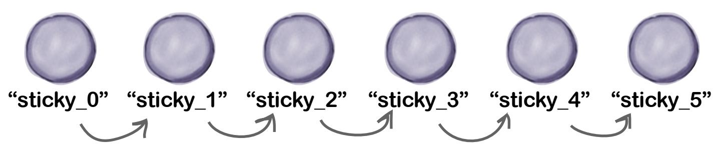
Five notes, labeled from zero to four.

To add a new sticky, we count the number of items in the local store and create our new key from that number:

```
var key = "sticky_" + localStorage.length;
```



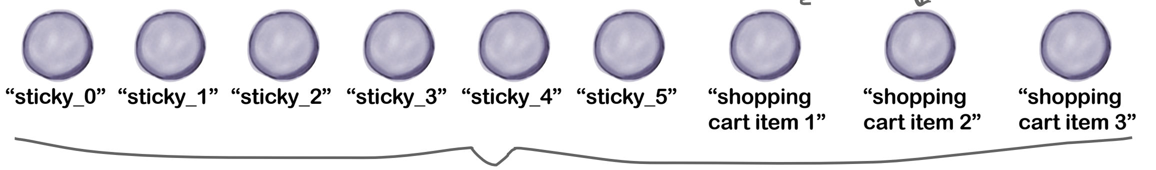
And to display all the stickies, we iterate from zero to the length of the local store (minus one):



Length is now six, so iterate zero to five, displaying each note from "sticky_0" to "sticky_5".

Now let's add Joel's items from his shopping cart to localStorage:

Here are Joel's items that he's using in his shopping cart code.



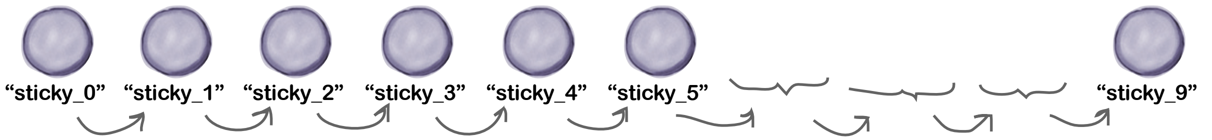
Now we have nine total items in localStorage.

And let's create a new sticky:

```
var key = "sticky_" + localStorage.length;
```

When we create our new sticky, the local store's length is now nine, so we create a note named "sticky_9". Hmm, that doesn't seem right.

When we need to iterate through the stickies to display them, we're in trouble:



Length is now ten (we just added a new sticky), so iterate zero to nine, displaying each sticky from "sticky_0" to "sticky_9".

Uh oh, there's no "sticky_6", "sticky_7" or "sticky_8".

Sharpen your pencil

Put a check next to the ways our current implementation could cause problems:

- [ ] Displaying stickies is inefficient if there are a lot of items in localStorage that aren't stickies.
- [ ] A sticky could be overwritten by setItem if the size of the localStorage gets smaller when another app deletes its own items.
- [ ] It's hard to quickly tell how many stickies there are; you have to iterate through every item in localStorage to get all the stickies.
- [ ] Use a cookie, it has to be easier than all this!


If only I could store an array in localStorage. We could use it to hold all the keys of the stickies and we could also always easily know the number of stickies we're storing. But we all know localStorage stores only strings, so even though an array would be dreamy, I know it's just a fantasy...

We have the technology...

We haven't been lying, it is true that you can store only strings as the values of localStorage items, however that isn't the whole truth because we can always convert an array (or an object) into a string before we store it. Sure, it seems like cheating, but it's a totally legit way to store your non-String data types in localStorage.

We know you're dying to jump into the nitty-gritty of how to store arrays, but before we do, let's first step through how an array would actually solve our (and Joel's) problems.

Let's rewind and say we've got six stickies in localStorage:

Six stickies, labeled from zero to five.

The stickies and the stickies array are both stored in localStorage.

and we've got an array in localStorage named "stickiesArray":

"sticky_0"	"sticky_1"	"sticky_2"	"sticky_3"	"sticky_4"	"sticky_5"

"stickiesArray"

Each element of the stickies array is a key to a sticky in localStorage.

Now let's add a new sticky. Let's call the sticky "sticky_815". Why such a crazy number? Because we're not going to care what it is called anymore as long as it is unique. So, to add the sticky, we just add "sticky_815" to the array and then store an item for the sticky, just like we have been. Like this:

We've got an extra sticky in localStorage.

Seven stickies: their keys no longer matter, they just need to be unique.

"sticky_0"	"sticky_1"	"sticky_2"	"sticky_3"	"sticky_4"	"sticky_5"	"sticky_815"

"stickiesArray"

And we've extended the stickies array by one value.

Reworking our app to use an array

Okay, we know roughly how we're going to keep track of our stickies using an array, but let's take this a little further and make sure we can iterate through and display all the stickies. In the current code we display all the stickies in the `init` function. Can we rewrite that using an array? We'll look at the existing code first, and then see how it changes (hopefully for the better) with an array. Don't type in this code yet; we're focusing on the changes we need to make for now and not making this code bulletproof. We'll bring on the bulletproof stuff in just a bit.

Before...

```
function init() {
    // button code here...
    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        if (key.substr(0, 6) == "sticky") {
            var value = localStorage.getItem(key);
            addStickyToDOM(value);
        }
    }
}
```

Here's our old code that relies on the stickies having specific names, sticky_0, sticky_1, and so on..

Wow, this was messy, come to think of it.

As we now know, this might break because we can't depend on all stickies to be there if we're naming them based on the count of the items in localStorage.

New and improved

```
function init() {
    // button code here...
    var stickiesArray = localStorage["stickiesArray"];
    if (!stickiesArray) {
        stickiesArray = [];
        localStorage.setItem("stickiesArray", stickiesArray);
    }

    for (var i = 0; i < stickiesArray.length; i++) {
        var key = stickiesArray[i];
        var value = localStorage[key];
        addStickyToDOM(value);
    }
}
```

We're starting by grabbing the stickiesArray out of localStorage.

We need to make sure there is an array in localStorage. If there isn't one, then let's create an empty one.

We're iterating here through the array.

Each element of the array is the key of a sticky, so we're using that to retrieve the corresponding item from localStorage.

And then we add that value to the DOM just like we have been.

NOTE: you still don't know how to store and retrieve arrays in localStorage, so treat this as pseudo-code until we show you. We'll have to make a very small addition for this to work.

Exercise

We still need to figure out how to actually store an array in localStorage.

You might have already guessed that we can use JSON to create a string representation of an array and if so, you're right. And once you have that you can store it in localStorage.

Recall that there are only two methods in the JSON API: stringify and parse. Let's put these methods to work by finishing the init function (check the solution at the end of the chapter before moving on):

```
function init() {
    // button code here...
    var stickiesArray = localStorage["stickiesArray"];
    if (!stickiesArray) {
        stickiesArray = [];
        localStorage.setItem("stickiesArray", _______________(stickiesArray));
    } else {
        stickiesArray = ___________ (stickiesArray);
    }
    for (var i = 0; i < stickiesArray.length; i++) {
        var key = stickiesArray[i];
        var value = localStorage[key];
        addStickyToDOM(value);
    }
}
```

We added this else clause because you'll need to do something if you get the array from localStorage (because it's a string not an array).

Converting createSticky to use an array

We've almost got this app covered. All we need to do is to rework the `createSticky` method, which, as you'll remember, just gets the text for the sticky from the form, stores it locally, and then displays it. Let's look at the curent implementation before changing it:

```
function createSticky() {
    var value = document.getElementById("note_text").value;
    var key = "sticky_" + localStorage.length;
    localStorage.setItem(key, value);

    addStickyToDOM(value);
}
```

Rather than using the localStorage length to create a key, which we've seen can cause problems, we're going to need to create a more unique key.

We're also going to need to add the sticky to our stickies array and save the array in localStorage.

What needs to change?

We have two things that need to change in `createSticky`. First, we need a new way to generate a key for each sticky that is unique. We also need to alter the code so that it stores the sticky in the `stickiesArray` in localStorage.

1 **We need to create a unique key for the sticky**

There are lots of ways to create unique keys. We could use the date and time, or create fancy random 64-bit numbers, or hook our app up to an atomic-clock API. Hmm, the date and time sounds like a nice, easy way to do this. JavaScript supports a date object that returns the number of milliseconds since 1970; that should be unique enough (unless you're going to create your stickies at a *really fast* rate):

Create a Date object, then get the current time in milliseconds.

```
var currentDate = new Date();
var time = currentDate.getTime();
var key = "sticky_" + time;
```

Our new code to create a unique key.

And then create the key by appending the milliseconds to the string "sticky_".

2 **We need to store the new sticky in the array**

Now that we have a way to create a unique key, we need to store the text of the sticky with that key, and add the key to the `stickiesArray`. Let's work through how to do that, and then we'll put all this code together.

Let's first grab the stickies array.

```
var stickiesArray = getStickiesArray();
localStorage.setItem(key, value);
stickiesArray.push(key);
localStorage.setItem("stickiesArray",
                     JSON.stringify(stickiesArray));
```

Rather than repeat all that code to get and check the stickiesArray, just like we did in init (on the previous page), we're going to create a new function to do it. We'll get to this in just a sec.

We then store the key with its value like we always did (only with our new key).

We then use the array method push, which appends the key onto the end of the stickies array.

And store the array back in localStorage, stringifying it first.

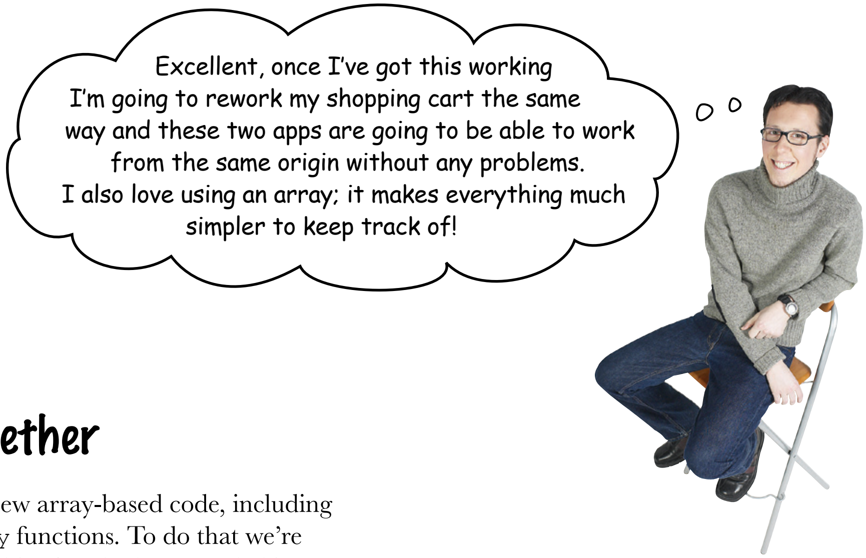
there are no Dumb Questions

Q: What are milliseconds since 1970?

A: You might already know a millisecond is a 1000th of a second, and the `getTime` method returns a count of milliseconds that have occurred since 1970. Why 1970? That behavior is inherited from the Unix operating system, which defined time that way. While it isn't perfect (for instance, it represents times before 1970 with negative numbers), it does come in handy when you need a unique number or to track time in JavaScript code.

Q: Isn't all this parsing and stringifying of JSON types rather inefficient? And if my array gets really large isn't that also going to be inefficient to store?

A: Theorectially yes on both counts. But for typical web page programming tasks it usually isn't an issue. That said, if you're implementing a serious application with very large storage requirements, you could see issues using JSON to convert items to and from strings.



Putting it all together

It's time to integrate all this new array-based code, including the `init` and `createSticky` functions. To do that we're first going to abstract a small bit of code that's needed in both functions—it's the code that retrieves the stickies array from localStorage. You've seen it in `init`, and we need it again in `createSticky`. Let's take that code and put it in a method called `getStickiesArray`—it should look familar to you given the code we've already walked through:

First we get the item "stickiesArray" out of localStorage.

```
function getStickiesArray() {

    var stickiesArray = localStorage.getItem("stickiesArray");

    if (!stickiesArray) {

        stickiesArray = [];

        localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));

    } else {

        stickiesArray = JSON.parse(stickiesArray);

    }

    return stickiesArray;

}
```

If this is the first time we've loaded this app, there might not be a "stickiesArray" item.

And if there isn't an array yet we create an empty array, and then store it back in localStorage.

Don't forget to stringify it first!

Otherwise, we found the array in localStorage, and we need to parse it to convert it to a JavaScript array.

In either case, we end up with an array, and we return it.

Putting it all together continued...

With `getStickiesArray` written, let's look at the simplified, final versions of the `init` and `createSticky` functions. Go ahead and type these in:

```
function init() {
    var button = document.getElementById("add_button");
    button.onclick = createSticky;

    var stickiesArray = getStickiesArray();

    for (var i = 0; i < stickiesArray.length; i++) {
        var key = stickiesArray[i];
        var value = localStorage[key];
        addStickyToDOM(value);
    }
}
```

Remember we also set up the button events here in the init method.

Next we grab the array with the stickies' keys in it.

Now we're going to iterate through the stickies array (not the localStorage items!).

Each item in the array is a key to a sticky. Let's grab each one.

And grab its value from localStorage.

And add it to the DOM just like we've been doing.

With `init` finished, we just have `createSticky` left:

```
function createSticky() {
    var stickiesArray = getStickiesArray();
    var currentDate = new Date();
    var key = "sticky_" + currentDate.getTime();
    var value = document.getElementById("note_text").value;
    localStorage.setItem(key, value);
    stickiesArray.push(key);
    localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));
    addStickyToDOM(value);
}
```

We start by grabbing the stickies array.

Then let's create that unique key for our new sticky.

We add sticky key/value to localStorage.

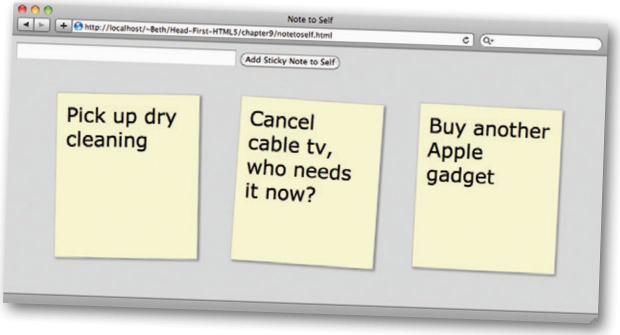
And add the new key to the stickies array...

And then we stringify the array and write back to localStorage.

Finally, we update the page with the new sticky by adding the sticky to the DOM.

Test Drive!

Get all this code in and clear out your localStorage to make a nice clean start. Load this code, and you should see exactly the same behavior as last time. Joel, you'll see your code working correctly now!



there are no Dumb Questions

Q: We're using "sticky_" as the prefix for our localStorage item names. Is there a convention for localStorage naming schemes?

A: There is no convention for naming localStorage items. If your web app is on a small site at a domain that you have control over, then naming shouldn't be an issue since you'll be aware of all the names being used by all the different pages at the site. We think it's probably a good idea to use a name that indicates the page or web app relying on that item. So "sticky_" helps us remember that those items are related to the Sticky Notes app.

Q: So if my sticky notes app is just one of many apps at a domain, I have to worry about potential conflicts right?

A: Yes. In that case, it would be a good idea for you (or someone who manages the web sites at the domain) to put together a plan for how to name items.

Q: If I have a lot of stickies, my stickiesArray is going to get very long. Is that a problem?

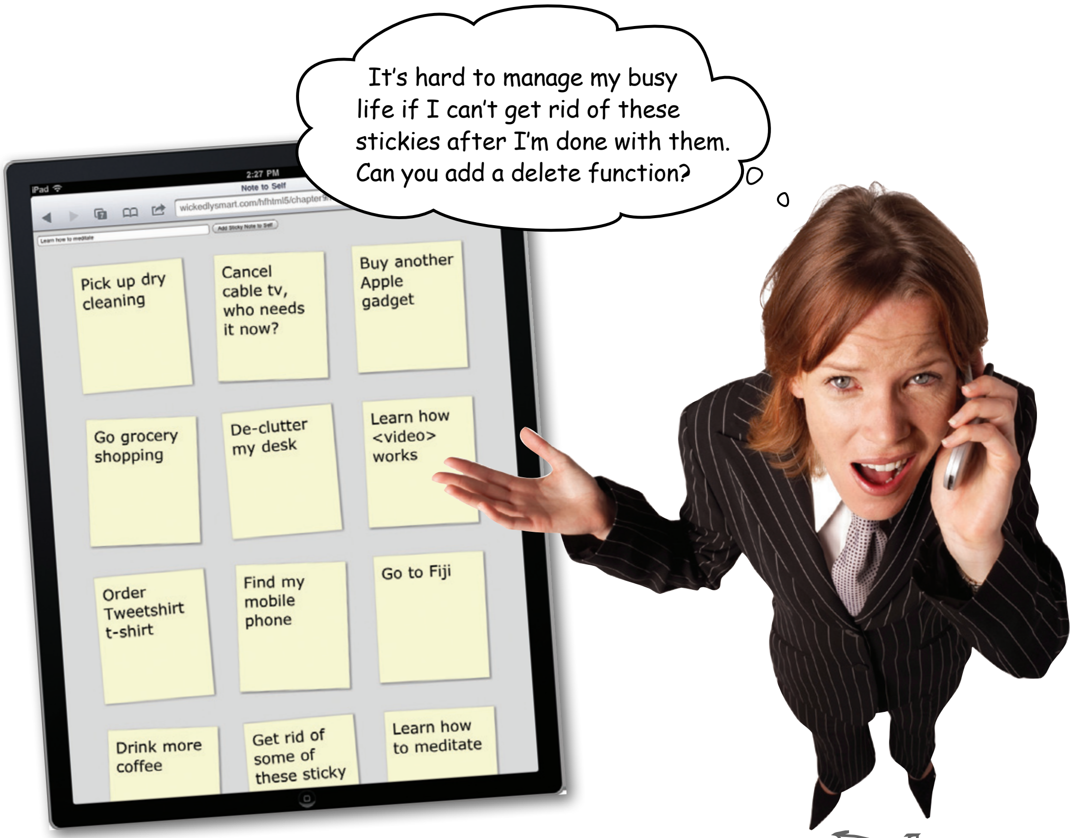
A: Unless you create thousands of stickies, it shouldn't be (and if you do create thousands of stickies, we want to know how you are so productive!). JavaScript is pretty fast these days.

Q: So just to be clear, we can store any object in localStorage, just by stringifying it first with JSON?

A: Right. JSON strings are simplified versions of JavaScript objects, and most simple JavaScript objects can be turned into a string using JSON and stored in localStorage. That includes arrays (as you've seen) as well as objects containing property names and values, as you'll see shortly.

Pick a naming scheme for your localStorage items that won't conflict with those of other applications at the same domain.

If you need to store arrays or objects in localStorage, use JSON.stringify to create the value to store, and JSON.parse after you retrieve it.



Be careful around the sharp objects!

Deleting sticky notes

She's right, this app isn't going to be very successful if we can't remove stickies. We've already mentioned the `localStorage.removeItem` method in this chapter, but we haven't really talked about it. The `removeItem` method takes the key of an item, and removes that item from localStorage:

```
localStorage.removeItem(key);
```

This method removes the item in localStorage with the given key.

removeItem has one parameter: the key of the item to be removed.

That sounds easy enough, doesn't it? Ah, but if you think about it, there is more to removing an sticky note than calling the `removeItem` method—we also need to deal with `stickiesArray`...

Sharpen your pencil

Let’s delete a sticky!

Below you’ll see the contents of localStorage. You’ve got all the JavaScript you want along with the removeItem method. Using a pencil, sketch out what you need to do to remove `sticky_1304220006342` from localStorage. After you’ve sketched it out, go ahead and write some pseudo-code below to show how you’re going to write your code.

“sticky_1304294652202”	“sticky_1304220006342”	“sticky_1304221742310”	“sticky_1304221683892”

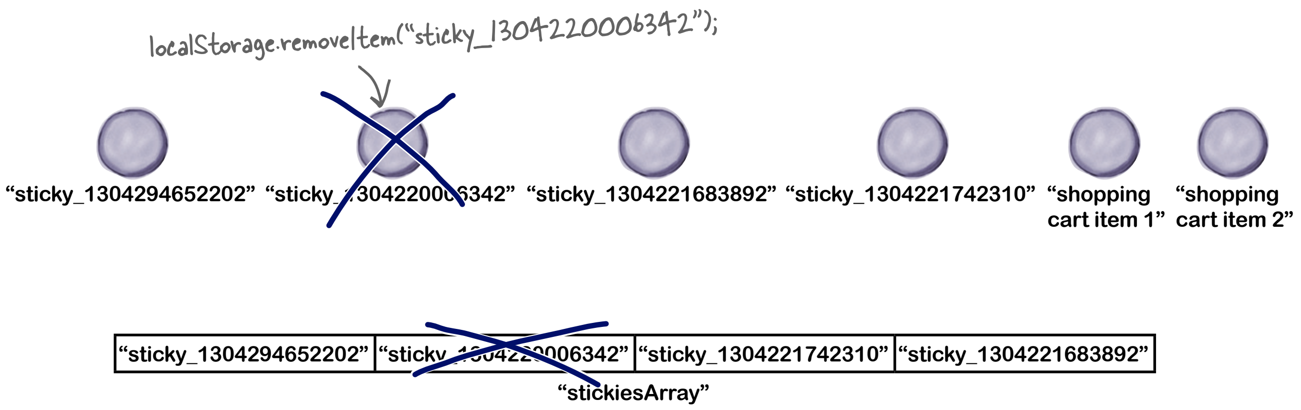
“stickiesArray”

Your pseudo-code here

Sharpen your pencil Solution

Let's delete a sticky!

Below you'll see the contents of localStorage. You've got all the JavaScript you want along with the removeItem method. Using a pencil, sketch out what you need to do to remove `sticky_1304220006342` from localStorage. After you've sketched it out, go ahead and write some pseudo-code below to show how you're going to write your code. Here's our solution.



(1) Remove the sticky with the key "sticky_1304220006342" from localStorage using the localStorage.removeItem method.

(2) Get the stickiesArray.

(3) Remove element with key="sticky_1304220006342" from the stickiesArray.

(4) Write stickiesArray back into localStorage (stringifying it first).

(5) Find "sticky_1304220006342" in the DOM and remove it.

The deleteSticky function

You made a plan for how to delete the sticky notes, so let's take a look at the `deleteSticky` function:

First, we remove the sticky note from localStorage using removeItem, passing in the key of the sticky to delete.

We're using the getStickiesArray function to get the stickiesArray from localStorage.

```
function deleteSticky(key) {
    localStorage.removeItem(key);
    var stickiesArray = getStickiesArray();
    if (stickiesArray) {
        for (var i = 0; i < stickiesArray.length; i++) {
            if (key == stickiesArray[i]) {
                stickiesArray.splice(i,1);
            }
        }

        localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));
    }
}
```

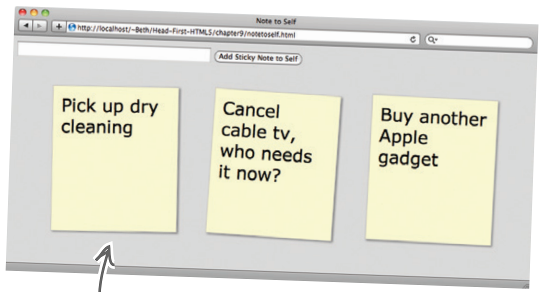
We make sure we have a stickiesArray (just in case), and then iterate through the array looking for the key we want to delete.

When we find the right key, we delete it from the array using splice.

splice removes elements from an array starting at the location given by the first argument (i), for as many elements as are specified in the second argument (1).

Finally, we save the stickiesArray (with the key removed) back to localStorage.

How do you select a sticky to delete?



When we click on a sticky note, it will get deleted.

We need a way for the user to select a sticky note to delete. We could get all fancy and add a little delete icon to each note, but for our Sticky Notes app, we're going to do something much simpler: we're going to just delete the sticky note if the user clicks on it. That may not be the best implementation in terms of usability, but it's straightforward.

To implement this, we first need to change the stickies so that we can detect *when* a sticky is clicked on, and then we'll pass that along to the `deleteSticky` function.

A lot of this needs to happen in the `addStickyToDOM` function, let's see how:

Big picture: we're going to use the key of the sticky note, which, remember, is "sticky_" + time, to uniquely identify the note. We'll pass in this key whenever we call addStickyToDOM.

```
function addStickyToDOM(key, value) {
    var stickies = document.getElementById("stickies");
    var sticky = document.createElement("li");
    sticky.setAttribute("id", key);
    var span = document.createElement("span");
    span.setAttribute("class", "sticky");
    span.innerHTML = stickyObj.value;
    sticky.appendChild(span);
    stickies.appendChild(sticky);
    sticky.onclick = deleteSticky;
}
```

We're adding a unique id to the <li> element that represents the sticky in the DOM. We're doing this so deleteSticky will know which sticky you clicked on. Since we already know the sticky's key is unique, we're just using that as the id.

We're also adding click handler to every sticky. When you click on a sticky, deleteSticky will be called.

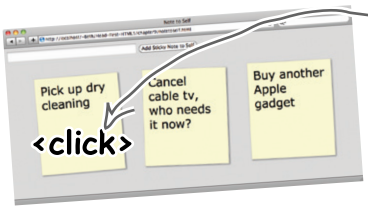
Exercise

Your job now is to update all the code so that everywhere we're calling addStickyToDOM, we're passing in the key as well as the value. You should be able to easily find these places. But after you've finished , check the solution at the end of the chapter to make sure.

Don't skip this, or the upcoming test drive won't work!

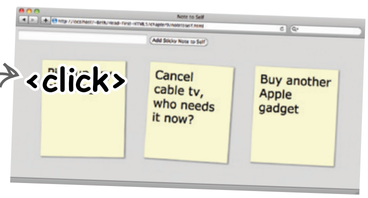
How to get the sticky to delete from the event

We've now got an event handler on each sticky note listening for clicks. When you click on a sticky, `deleteSticky` will be called and an event object will be passed into `deleteSticky` with information about the event, like which element was clicked on. We can look at the `event.target` to tell which sticky was clicked on. Let's take a closer look at what happens when you click on a sticky note.



If you click on the yellow part of a sticky, the event target is the <li> element. This is what we want, because the <li> now has an id with the key of the sticky note.

If you click on the text, the event target is the <span> inside the <li>, which is not what we want.




```
<li id="sticky_1304270008375">
     <span class="sticky">Pick up dry cleaning</span>
</li>
```

This is the HTML for the sticky note that we create in addStickyToDOM.

Either way, the event generated by your click gets passed into deleteSticky.

The target is the element you clicked on that generated the event, and we can get the id of that element from the target property. If the target is <li>, we're set.

```
function deleteSticky(e) {
    var key = e.target.id;
    if (e.target.tagName.toLowerCase() == "span") {
        key = e.target.parentNode.id;
    }
    localStorage.removeItem(key);
    var stickiesArray = getStickiesArray();
    if (stickiesArray) {
        for (var i = 0; i < stickiesArray.length; i++) {
            if (key == stickiesArray[i]) {
                stickiesArray.splice(i,1);
            }
        }
        localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));
        removeStickyFromDOM(key);
    }
}
```

If the target is the <span>, then we need to get the id of the parent element, the <li>. The <li> is the element with the id that is the key we need.

Now we can use the key to remove the item from localStorage, and from the stickiesArray.

We also need to remove the <li> holding the sticky from the page, so it disappears when you click it. We'll do that next...

Delete the sticky from the DOM, too

To finish up the delete, we need to implement the `removeStickyFromDOM` function. You updated the `addStickyToDOM` function earlier to add the key of the sticky as the id of the `<li>` element holding the sticky in the DOM, so we can use `document.getElementById` to find the sticky in the DOM. We get the parent node of the sticky, and use the `removeChild` method to delete the sticky:

Pass in the key (also the id) of the sticky element we're looking for.

```
function removeStickyFromDOM(key) {
    var sticky = document.getElementById(key);
    sticky.parentNode.removeChild(sticky);
}
```

  remove the child node

We grab the <li> element from the DOM...

... and remove it by first getting its parentNode and then using removeChild to remove it.

Okay, test it... 

Get all that code in, load the page, add and delete some stickies. Quit your browser, load it again, and give it a real run through!

We can delete stickies now!

Nice work! Now, can you give me a way to color code my stickies? You know yellow for urgent, blue for ideas, pink for backburner, that kind of thing?

But of course we can!

Come on, given your level of experience with this we're going to be able to knock this out. How do we do it? Well, we're going to create an object to store the text of the note and its color, and then we're going to store that as the value of the sticky item, using `JSON.stringify` to convert it to a string first.

Update the user interface so we can specify a color

Right now, all our notes are yellow. Wouldn't it be nicer if we could have a whole range of sticky note colors?

We could add a selection menu up here so you can choose the color you want for your note.

This is way better, don't you think?

Let's tackle the easy part first: updating the HTML so we have a selection menu of colors to choose from. Edit your `notetoself.html` file and update your form to add the colors like this:

```
<html>
...
    <form>
        <label for="note_color">Color: </label>
        <select id="note_color">
            <option value="LightGoldenRodYellow">yellow</option>
            <option value="PaleGreen">green</option>
            <option value="LightPink">pink</option>
            <option value="LightBlue">blue</option>
        </select>
        <label for="note_text">Text:</label> <input type="text" id="note_text">
        <input type="button" id="add_button" value="Add Sticky Note to Self">
    </form>
...
</html>
```

We're only changing the form, the rest stays the same.

Notice the id of the <select>; we'll need that to grab the value of the selected option in the JavaScript.

We've added four sticky colors to choose from.

We'll add a label for the sticky text so the user knows what that field is for.

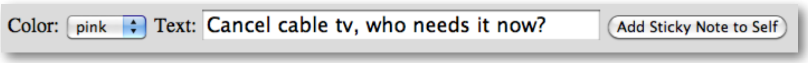
The value of each option is the name of a color we can just plug right into the style for our stickies.

And the rest of the form is the same.

We've been using CSS to define the default color for the notes. Now we want to store a note's color with the note itself. So, now the question is: how are we going to store the color for the sticky note in localStorage?

JSON.stringify, it's not just for Arrays

To store the color of the sticky with the text of the sticky, we can use the same technique we used for `stickiesArray`: we can store an object that contains the text and the color as the value for the sticky in localStorage.



We're going to take the values the user enters for the color and the sticky note text and package them up into a simple object.

And we'll store that in localStorage with the sticky's key.

localStorage

Just like stickiesArray, we'll have to call JSON.stringify on the sticky value before we call localStorage.setItem to save the value.

```
var stickyObj = {
    "value": "Cancel cable tv, who needs it now?",
    "color": "LightPink"
};
```

Let's rewrite the `createSticky` function to store the color with the sticky note text. To represent the text and the color, we'll use our handy object:

```
function createSticky() {
    var stickiesArray = getStickiesArray();
    var currentDate = new Date();
    var colorSelectObj = document.getElementById("note_color");
    var index = colorSelectObj.selectedIndex;
    var color = colorSelectObj[index].value;
    var key = "sticky_" + currentDate.getTime();
    var value = document.getElementById(“note_text”).value;
    var stickyObj = {
            "value": value,
            "color": color
    };
    localStorage.setItem(key, JSON.stringify(stickyObj));
    stickiesArray.push(key);
    localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));
    addStickyToDOM(key, stickyObj);
}
```

We do the usual thing to grab the value of the selected color option.

Then we use that color to create stickyObj: an object that contains two properties, the text of the sticky, and the color the user selected.

And, we JSON.stringify the stickyObj before we put it in localStorage.

Now, we're passing the object instead of a text string to addStickyToDOM. Which means you'll need to update addStickyToDOM too, right?

Using the new stickyObj

Now that we're passing stickyObj to `addStickyToDOM`, we need to update the function to use the object instead of the string we were passing in before, and to set the background color of the sticky. It's a fairly easy change though; let's take a look:

```
function addStickyToDOM(key, stickyObj) {
    var stickies = document.getElementById("stickies");
    var sticky = document.createElement("li");
    sticky.setAttribute("id", key);

    sticky.style.backgroundColor = stickyObj.color;

    var span = document.createElement("span");
    span.setAttribute("class", "sticky");
    span.innerHTML = stickyObj.value;
    sticky.appendChild(span);
    stickies.appendChild(sticky);
    sticky.onclick = deleteSticky;
}
```

We need to change the parameter here to be the stickyObj rather than the text value of the sticky.

We get the color from the stickyObj we're passing into addStickyToDOM.

Notice that when we set the background color property in JavaScript, we specify it as backgroundColor, NOT background-color, like in CSS.

HTML element objects have a style property you can use to access the style of that element.

And then we need to get the text value we're going to use in the sticky note from the object.

There is one other place we need to update the code, and that is in `init`, where we are getting the stickies from localStorage and passing to `addStickyToDOM` when we first load the page.

```
function init() {
    var button = document.getElementById("add_button");
    button.onclick = createSticky;

    var stickiesAray = getStickiesArray();

    for (var i = 0; i < stickiesArray.length; i++) {
        var key = stickiesArray[i];
        var value = JSON.parse(localStorage[key]);
        addStickyToDOM(key, value);
    }
}
```

Now when we get the value of the sticky note from localStorage, we need to JSON.parse it, because it's an object, not a string anymore.

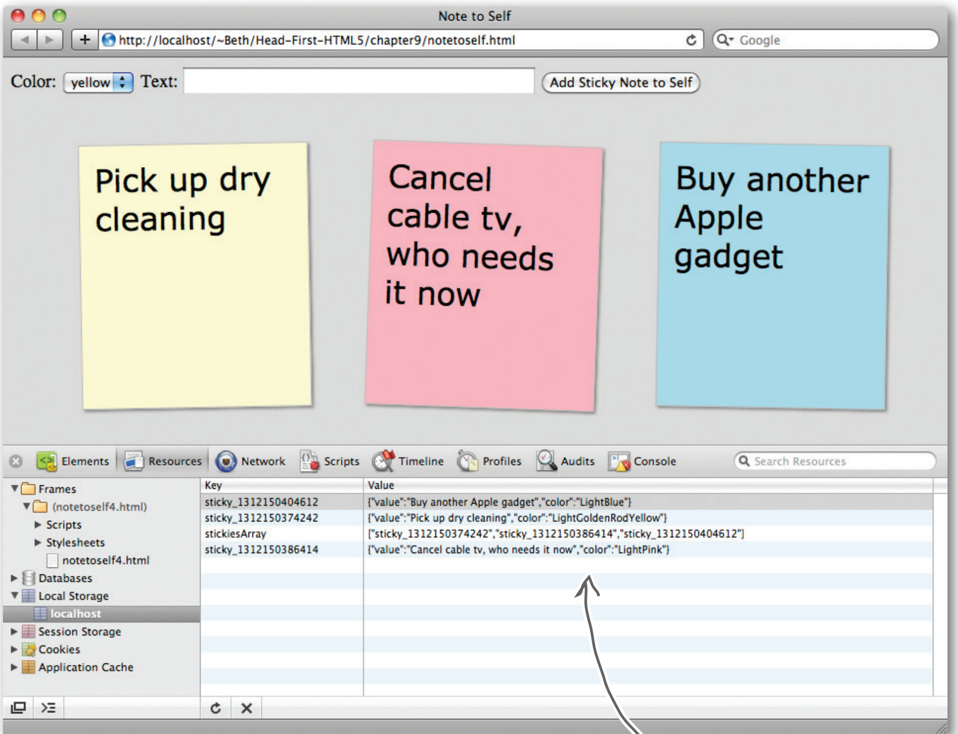
And we pass that object to addStickyToDOM instead of the string (the code looks the same, but the thing we're passing is different).

Test drive sticky note colors

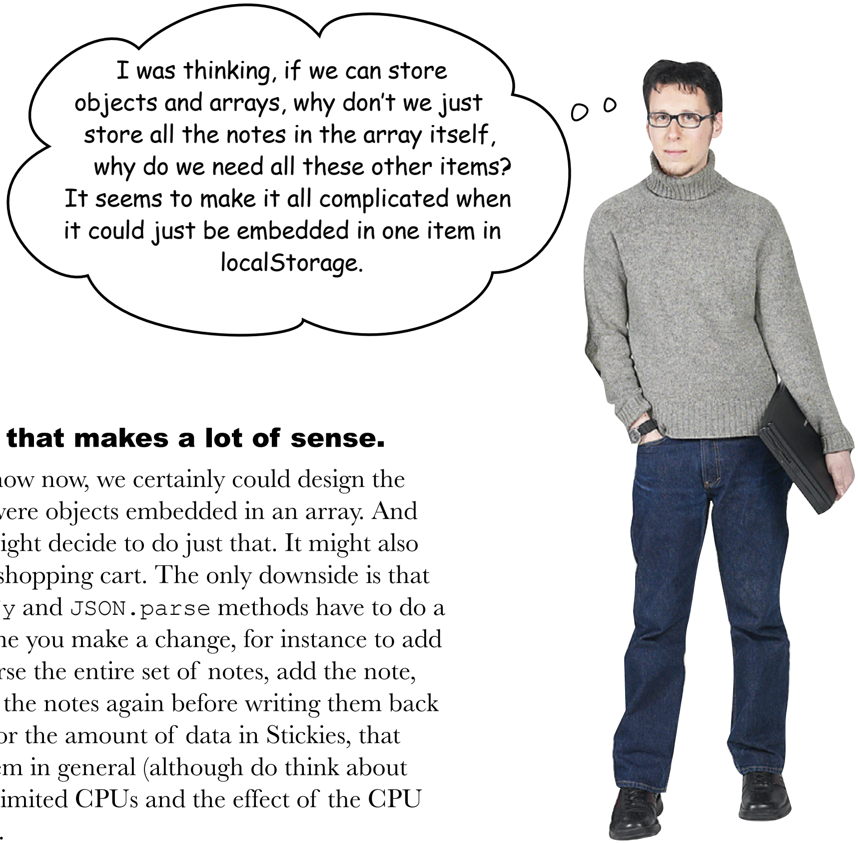
Before running the Note to Self app again, you'll need to clear out your localStorage first because the previous version of our stickies didn't have any color stored in them, and now we're using a different format for our sticky values. Before we were using strings, now we're using objects. So empty out your localStorage, reload the page, and add some stickies, selecting a different color for each one. Here are our stickies (and we'll also check out localStorage too):

You can use your maintenance.html file to clear out your localStorage, or use the console.

We picked yellow, pink, and blue for our sticky notes when we added them.



Each sticky note's value is now a (JSON stringified) object containing the text value of the sticky and the color of the sticky.



For some uses, that makes a lot of sense.

Knowing what we know now, we certainly could design the stickies so that they were objects embedded in an array. And going forward you might decide to do just that. It might also make sense for your shopping cart. The only downside is that the `JSON.stringify` and `JSON.parse` methods have to do a lot more work anytime you make a change, for instance to add a note we have to parse the entire set of notes, add the note, and then stringify all the notes again before writing them back in to the store. But, for the amount of data in Stickies, that shouldn't be a problem in general (although do think about mobile devices with limited CPUs and the effect of the CPU usage on battery life).

So whether you want to pack everything into one object or array in localStorage, really depends on how many data items you need to store, how big each one is, and what type of processing you're going to do on them.

While our implementation here may be a bit of overkill for a limited number of Stickies, we hope you agree it gave us a great way to think about the localStorage API and how to deal with items in it.

~~DON'T~~ TRY THIS AT HOME
(OR BLOWING UP YOUR 5 MEGABYTES)

We've told you that you have five whole megabytes of storage on every user's browser, but while five megabytes sounds like a lot, remember that all your data is stored in the form of a string rather than in a byte-efficient data format. Take a long number, say, the national debt—when expressed in floating point form it takes up very little storage, but when expressed in the form of a string, it takes up many times that amount of memory. So, given that, the five megabytes might not hold as much as you think.

So what happens when you use all 5MBs? Well, unfortunately this is one of those behaviors that isn't well defined by the HTML5 specification, and browsers may do different things when you exceed your limit—the browser may ask if you want to allow more storage, or it may throw a `QUOTA_EXCEEDED_ERR` exception, which you can catch like this:

Here's a setItem call in the middle of the try block; if anything goes wrong and setItem throws an exception, the catch block will be invoked.

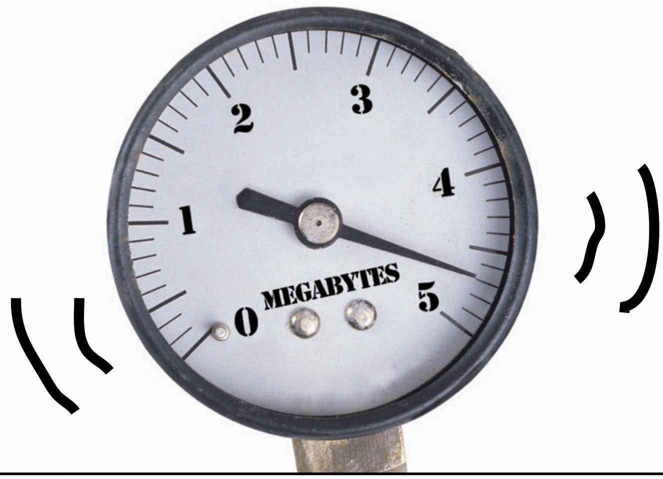
A try/catch captures any exceptions that are thrown within the try block.

```
try {
   localStorage.setItem(myKey, myValue);
} catch(e) {
   if (e == QUOTA_EXCEEDED_ERR) {
      alert("Out of storage!");
   }
}
```

We're testing to see if this is a storage quota error (as opposed to some other type of exception). If so, we alert the user. You'll most likely want to do something more meaningful than just an alert.

This is one JavaScript area we haven't covered, you might want to add it to your list of things to look into.

Not all browsers are currently throwing the QUOTA_EXCEEDED_ERR exception. But they still throw a exception when you exceed your limit, so you may want to handle the general case of an exception occuring when you set an item.



DANGER
Explosive Exercise

We don't see any reason not to push your browser to the limit, see what it's made of, see how far it can go, see what its behavior is under pressure. Let's write a little code to push your browser over its storage limit:

```
<html>
<head>
<script>

localStorage.setItem("fuse", "-");
while(true) {
    var fuse = localStorage.getItem("fuse");
    try {
        localStorage.setItem("fuse", fuse + fuse);
    } catch(e) {
        alert("Your browser blew up at" + fuse.length + " with exception: " + e);
        break;
    }
}
localStorage.removeItem("fuse");
</script>
</head>
<body>
</body>
</html>
```

Let start with a one-character string, with the key "fuse".

And just keep increasing its size...

...by doubling the string (by concatenating it with itself).

Then we'll try to write it back to localStorage.

If it blows up, we're done! We'll alert the user and get out of this loop.

And let's not leave a mess, so remove the item from localStorage.

Go ahead and type this in, light the fuse by loading it, and have fun! Try this on a few different browsers.

....

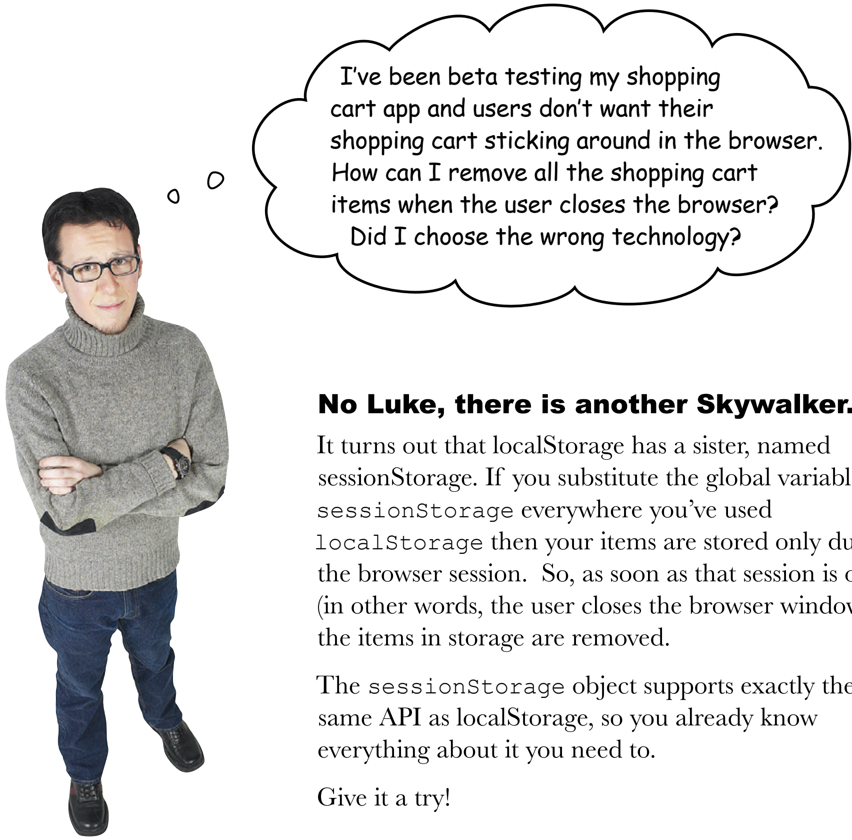
....

....

If you have the nerve to run this, put your results here.

Use at your own risk!

Seriously, this code could crash your browser, which might lead to your operating system being unhappy, which could lead to you losing work. Use at your own risk!!!



No Luke, there is another Skywalker.

It turns out that localStorage has a sister, named sessionStorage. If you substitute the global variable `sessionStorage` everywhere you've used `localStorage` then your items are stored only during the browser session. So, as soon as that session is over (in other words, the user closes the browser window), the items in storage are removed.

The `sessionStorage` object supports exactly the same API as localStorage, so you already know everything about it you need to.

Give it a try!

Who Does What?

At this point you've been through the localStorage API. Below you'll find all the main characters of the API sitting with their masks on. See if you can determine who does what. We've gone ahead and done one for you to get you started.

clear

sessionStorage

key

setItem

removeItem

length

getItem

localStorage

QUOTA_EXCEEDED_ERR

Use me to store items for the long term.

I take keys and values and write them into the localStorage. Now keep in mind if there's an item with that key already in the localStorage, I'm not going to warn you, I'm just going to overwrite it so you better know what you're asking for.

If you overstay your welcome in localStorage and use too much space you'll get an exception and you'll be hearing from me.

Need to knock off an item? I'll get the job done discreetly.

Just give me a key and I'll go out and find the item with that key and hand its value to you.

I'm a short term kinda guy, I'll store your stuff just as long as you have the browser open. Close your browser, and poof, all your stuff is gone.

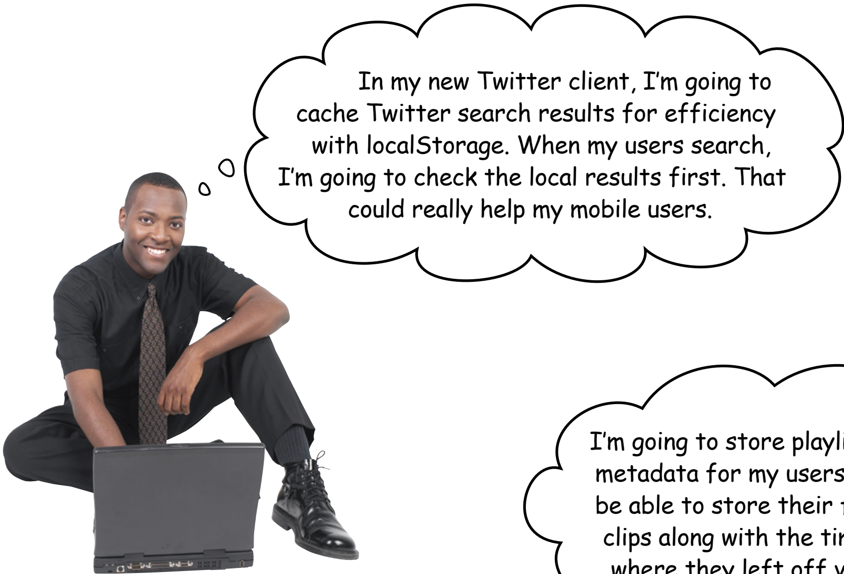
When you've had it with all the items in your localStorage, I clean up all those items and throw them away, leaving you with a nice fresh and empty localStorage (keep in mind I can only clean up my own origin).

Need to know how many items are in your localStorage? That's me.

Give me an index, and I'll give you a key from that index in localStorage.

Now that you know localStorage, how are you going to use it?

There are many ways to make use of localStorage—the Stickies app used them so we didn't need a server, but even with a server, localStorage can be quite helpful. Here's a few other ways developers are using them:




I've got a really cool game that works in two different browser windows, and I'm using localStorage to synchronize state.


I'm storing lots of local data to make my clients' apps fast on their mobile devices. Having a large store on the client side is a huge win for me.

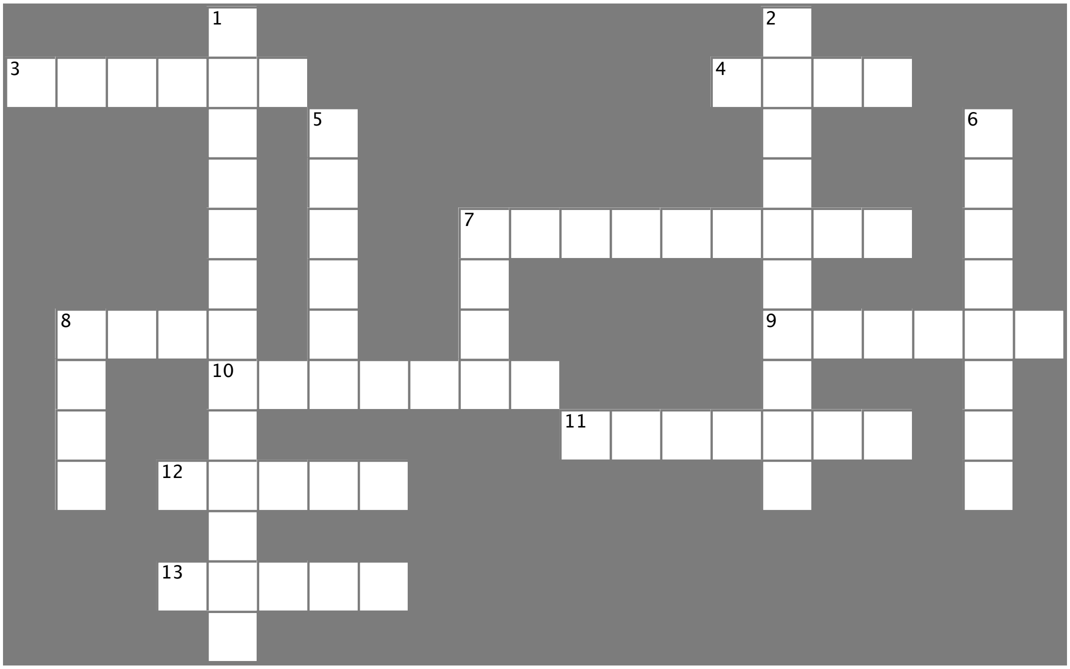

This gives me a new way to store user state. I used to need some kind of server-side based session and backend storage. Now I can just store my users' state locally, and bring in the server-side code only when I have to.

BULLET POINTS

- Web Storage is a store in your browser and an API you can use to save and retrieve items from the store.
- Most browsers provide at least 5 megabytes of storage per origin.
- Web Storage consists of local storage and session storage.
- Local storage is persistent, even if you close your browser window or quit the browser.
- Items in session storage are removed when you close your browser window or quit the browser. Session storage is good for temporary items, not longer term storage.
- Both local storage and session storage use exactly the same API.
- Web Storage is organized by origin (think domain). An origin is the location of the document on the Web (e.g., wickedlysmart.com or headfirstlabs.com).
- Each domain has a separate storage, so items stored in one origin are not visible to web pages in another origin.
- Use localStorage.setItem(key) to add a value to the store.
- Use localStorage.getItem(key) to retrieve a value from the store.
- You can use the same syntax as associative arrays to set and retrieve items to and from the store. Use localStorage[key] to do this.
- Use the localStorage.key() method to enumerate the keys in localStorage.
- localStorage.length is the number of items in localStorage at a given origin.
- Use the console in your browser to see and delete items in localStorage.
- You can delete items directly from localStorage by right-clicking on an item and choosing delete (note: may not work in all browsers).
- You can delete items from localStorage in code using the removeItem(key) method and the clear method. Note that the clear method deletes everything in localStorage at the origin where you do the clear.
- The keys for each localStorage item must be unique. If you use the same key as an existing item, you'll overwrite the value of that item.
- One way to generate a unique key is to use the current time in milliseconds since 1970, using the Date object's getTime() method.
- It is important to create a naming scheme for your web app that will still work if items are removed from the store, or if another app creates items in the store.
- Web Storage currently supports storing strings as values for keys.
- You can convert numbers stored in localStorage as strings back to numbers using parseInt or parseFloat.
- If you need to store more complex data, you can use JavaScript objects and convert them to strings before storing using JSON.stringify, and back to objects after retrieving using JSON.parse.
- Local storage may be particularly useful on mobile devices to reduce bandwidth requirements.
- Session storage is just like local storage, except that what's saved in the browser's store doesn't persist if you close the tab, the window, or exit the browser. Session storage is useful for short term storage, such as for a shopping session.

HTML5cross

Take some time to test your own local storage.



Across

3. When we used the _________ of localStorage to create key names, we ran into a problem: gaps in the names of our sticky notes.
4. Luke Skywalker's sister.
7. We have to _________ an object before we store it in localStorage.
8. Most browsers offer ________ megabytes of storage per origin.
9. We can detect which sticky note the user clicks on by looking at the event _________.
10. We store an item in localStorage with this method.
11. localStorage can store only _________________.
12. We thought it would be just a fantasy to store an _______ in localStorage but it turns out you can, with JSON.
13. Use a try/_______ to detect quota-exceeded errors in localStorage.

Down

1. We used the _____________ to hold the keys of all our stickies so we could easily find them in localStorage.
2. sessionStorage is just like localStorage except its not __________ if you close your browser window.
5. We create an _______ to store the sticky note text and its color in one localStorage item.
6. Use _________ to convert a string to an integer.
7. Cookie has a _________ issue.
8. If you store something in your browser and fly to ________ , it will still be there when you come back.

The Shell Game Solution

Ready to try your luck? Or should we say skill? We've got a game for you to test your command of localStorage, but you'll need to be on your toes. Use your knowledge of getting and setting key/value pairs in localStore to keep track of the pea as it shifts from shell to shell. Here's our solution.

```
function shellGame() {
    localStorage.setItem("shell1", "pea");
    localStorage.setItem("shell2", "empty");
    localStorage.setItem("shell3", "empty");
    localStorage["shell1"] = "empty";
    localStorage["shell2"] = "pea";
    localStorage["shell3"] = "empty";
    var value = localStorage.getItem("shell2");
    localStorage.setItem("shell1", value);
    value = localStorage.getItem("shell3");
    localStorage["shell2"] = value;
    var key = "shell2";
    localStorage[key] = "pea";
    key = "shell1";
    localStorage[key] = "empty";
    key = "shell3";
    localStorage[key] = "empty";

    for (var i = 0; i < localStorage.length; i++) {
        var key = localStorage.key(i);
        var value = localStorage.getItem(key);
        alert(key + ": " + value);
    }
}
```

Which shell had the pea?

Key	Value
shell1	empty
shell2	pea
shell3	empty

The pea is under shell2.

Your job was to update all the code so that everywhere we're calling addStickyToDOM, we're passing in the key as well as the value.

You should have updated all the calls to addStickyToDom in `init` and `createSticky`, to look like this:

```
addStickyToDOM(key, value);
```

Sharpen your pencil Solution

Put a check next to the ways our current implementation could cause problems:

- [x] Displaying stickies is inefficient if there are a lot of items in localStorage that aren't stickies.
- [x] A sticky could be overwritten by setItem if the size of the localStorage gets smaller when another app deletes its own items.
- [x] It's hard to quickly tell how many stickies there are; you have to iterate through every item in localStorage to get all the stickies.
- [ ] Use a cookie, it has to be easier than all this!

Exercise Solution

We still need to figure out how to actually store an array in localStorage

You might have already guessed that we can use JSON to create a string representation of an array and if so, you're right. And once you have that you can store it in localStorage.

Recall that there are only two methods in the JSON API, stringify and parse. Let's put those methods to work by finishing the init function:

Grab the array from localStorage.

If there isn't an array in localStorage, then we create an empty array and assign it to the variable stickiesArray. At this point, the variable stickiesArray is a string.

If we had to create a new array, we use JSON. stringify to create a string representation of the array, and then we store it..

```
function init() {
    // button code here...
    var stickiesArray = localStorage["stickiesArray"];
    if (!stickiesArray) {
        stickiesArray = [];
        localStorage.setItem("stickiesArray", JSON.stringify(stickiesArray));
    } else {
        stickiesArray = JSON.parse(stickiesArray);
    }
    for (var i = 0; i < stickiesArray.length; i++) {
        var key = stickiesArray[i];
        var value = localStorage[key];
        addStickyToDOM(value);
    }
}
```

If the stickies array is already stored in localStorage (as a string), then we need to parse it using JSON. After this we'll have an array of keys assigned to the stickiesArray variable.

Just to be clear, we're taking the string pointed to by stickiesArray, parsing it into an array, and then assigning that array back to the stickiesArray variable.

~~DON~~'T TRY THIS AT HOME
(OR BLOWING UP YOUR 5 MEGABYTES)

We've told you that you have five whole megabytes of storage on every user's browser, but while five megabytes sounds like a lot, remember that all your data is stored in the form of a string rather than in a byte-efficient data format. Take a long number, say, the national debt—when expressed in floating point form it takes up very little storage, but when expressed in the form of a string, it takes up many times that amount of memory. So, given that, the five megabytes might not hold as much as you think.

So what happens when you use all 5MBs? Well, unfortunately this is one of those behaviors that isn't well defined by the HTML5 specification, and browsers may do different things when you exceed your limit—the browser may ask if you want to allow more storage, or it may throw a `QUOTA_EXCEEDED_ERR` exception, which you can catch like this:

A try/catch captures any exceptions that are thrown within the try block.

```
try {
   localStorage.setItem(myKey, myValue);
} catch(e) {
   if (e == QUOTA_EXCEEDED_ERR) {
      alert("Out of storage!");
   }
}
```

Here's a setItem call in the middle of the try block; if anything goes wrong and setItem throws an exception, the catch block will be invoked.

We're testing to see if this is a storage quota error (as opposed to some other type of exception). If so, we alert the user. You'll most likely want to do something more meaningful than just an alert.

Not all browsers are currently throwing the QUOTA_EXCEEDED_ERR exception. But they still throw a exception when you exceed your limit, so you may want to handle the general case of an exception occuring when you set an item.

We don't see any reason not to push your browser to the limit, see what it's made of, see how far it can go, see what its behavior is under pressure. Let's write a little code to push your browser over its storage limit:

```
<html>
<head>
<script>

localStorage.setItem("fuse", "-");
while(true) {
    var fuse = localStorage.getItem("fuse");
    try {
        localStorage.setItem("fuse", fuse + fuse);
    } catch(e) {
        alert("Your browser blew up at" + fuse.length + " with exception: " + e);
        break;
    }
}
localStorage.removeItem("fuse");
</script>
</head>
<body>
</body>
</html>
```

Let start with a one-character string, with the key "fuse".

And just keep increasing its size...

...by doubling the string (by concatenating it with itself).

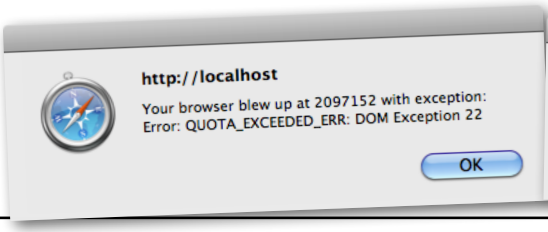
Then we'll try to write it back to localStorage.

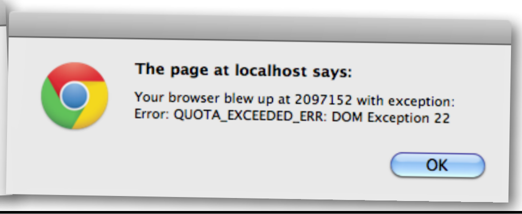
If it blows up, we're done! We'll alert the user and get out of this loop.

And let's not leave a mess, so remove the item from localStorage.

Go ahead and type this in, light the fuse by loading it, and have fun! Try this on a few different browsers.

Our results from Safari and Chrome.





Who Does What? Solution

At this point you've been through the localStorage API. Below you'll find all the main characters of the API sitting with their masks on. See if you can determine who does what. Here's our solution.

clear

sessionStorage

key

setItem

removeItem

length

getItem

localStorage

QUOTA_EXCEEDED_ERR

Use me to store items for the long term.

I take keys and values and write them into the localStorage. Now keep in mind if there's an item with that key already in the localStorage, I'm not going to warn you, I'm just going to overwrite it so you better know what you're asking for.

If you overstay your welcome in localStorage and use too much space you'll get an exception and you'll be hearing from me.

Need to knock off an item? I'll get the job done discreetly.

Just give me a key and I'll go out and find the item with that key and hand its value to you.

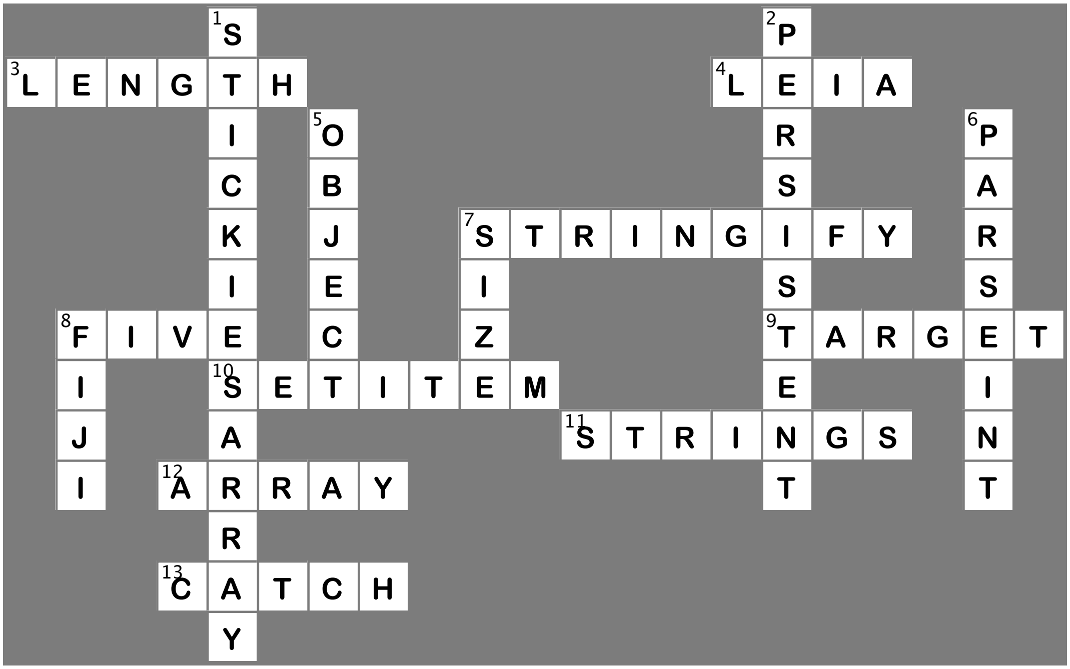
I'm a short term kinda guy, I'll store your stuff just as long as you have the browser open. Close your browser and, poof, all your stuff is gone.

When you've had it with all the items in your localStorage, I clean up all those items and throw them away, leaving you with a nice fresh and empty localStorage (keep in mind I can only clean up my own origin).

Need to know how many items are in your localStorage? That's me.

Give me an index, and I'll give you a key from that index in localStorage.

HTML5cross Solution



10 putting javascript to work

Web Workers

Slow script—do you want to continue running it?

If you've spent enough time with JavaScript or browsing the Web you've probably seen the "slow script" message. And, with all those multicore processors sitting in your new machine how could a script be running *too slow*? It's because JavaScript can only do one thing at a time. But, with HTML5 and Web Workers, *all that changes*. You've now got the ability to spawn *your own* JavaScript workers to get more work done. Whether you're just trying to design a more responsive app, or you just want to max out your machine's CPU, Web Workers are here to help. Put your JavaScript manager's hat on, let's get some workers cracking!

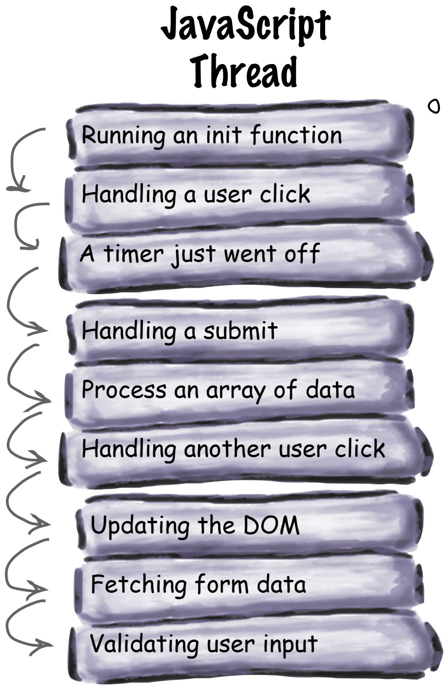
The Dreaded Slow Script

One of the great things about JavaScript is it does only one thing at a time. It's what we like to call "single-threaded." Why's that great? Because it makes programming straightforward. When you have lots of threads of execution happening at the same time, writing a program that works correctly can become quite a challenge.

The downside of being single-threaded is that if you give a JavaScript program too much to do, it can get overwhelmed, and we end up with "slow script" dialogs. The other ramification of having only one thread is if you have JavaScript code that is working really hard, it doesn't leave a lot of computational power for your user interface or your user's interactions, and your application can appear to be sluggish, or unresponsive.

How JavaScript spends its time

Let's see what this all means by taking a look at how JavaScript handles the tasks of a typical page:



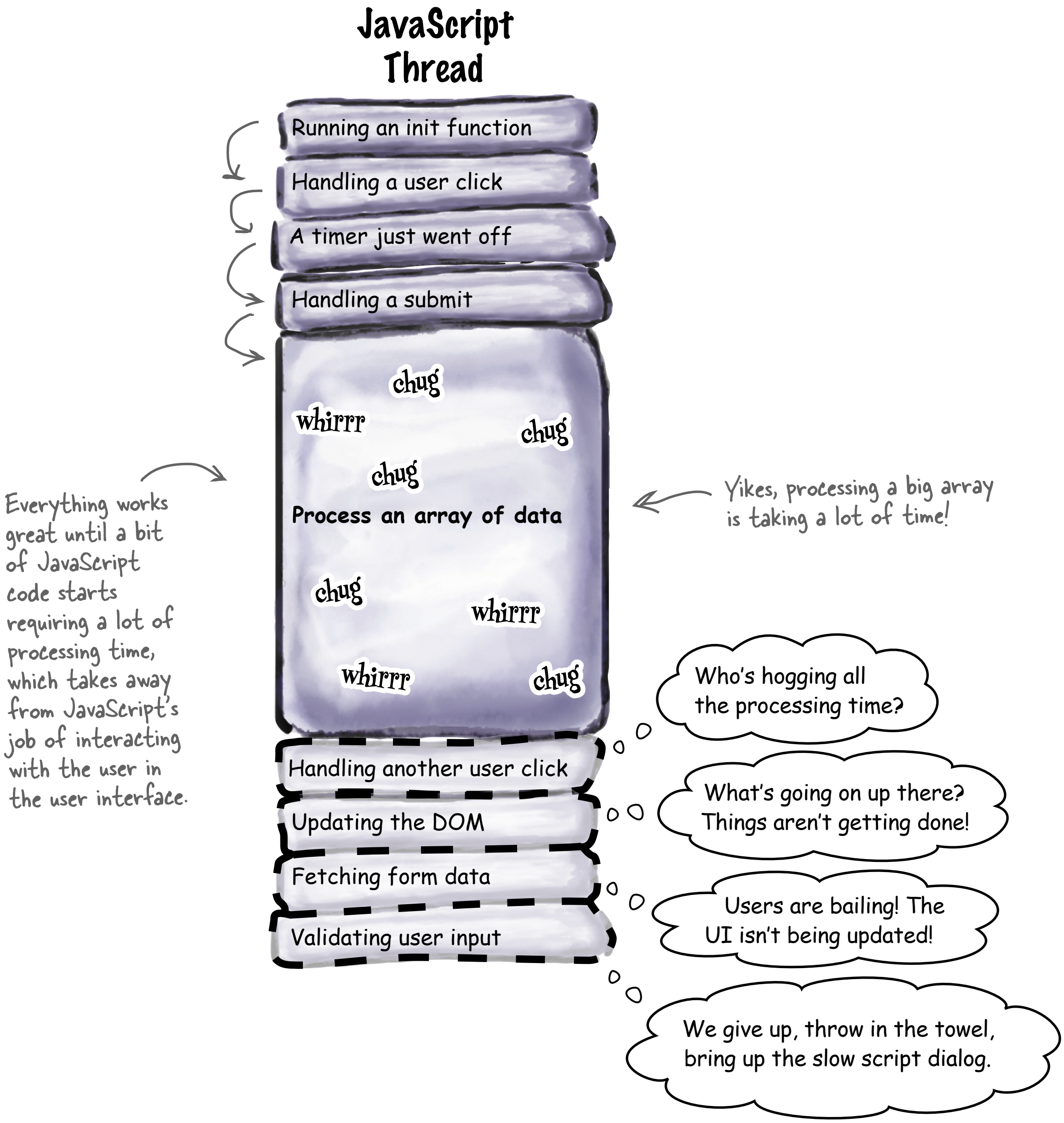
There's only one of me, but look at everything I get done by just handling all of this one thing at a time.

This is what we mean by single-threaded. JavaScript steps through everything it has to do, one after the other. There's no parallel execution going here.

For a lot of web apps this works really well. Everything gets done and the user interface seems fast and responsive.

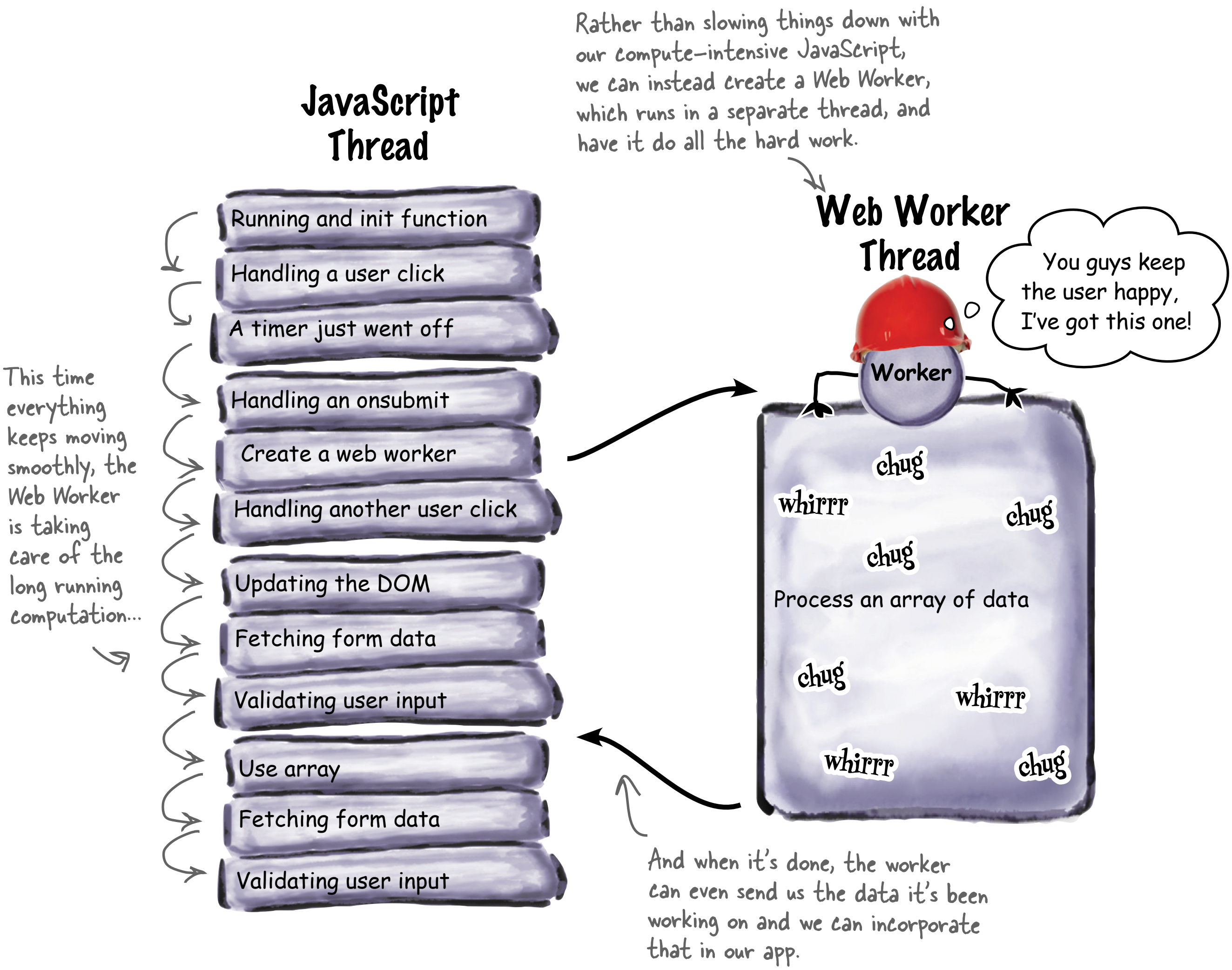
When single-threaded goes BAD

It's true, for a lot of uses, this single-threaded mode of computing by JavaScript works great, and as we've said, it makes programming straightforward. But, when you've written code that is so "computationally intensive" it starts to impact JavaScript's ability to get everything done, the single-threaded model starts to break down.



Adding another thread of control to help

Before HTML5, we were stuck with one thread of control in our pages and apps, but with Web Workers we've now got a way to create another thread of control to help out. So, if you've got code that takes a long time to compute, you can create a Web Worker that will handle that task while the main JavaScript thread of control is making sure everything is good with the browser and the user.



Now, we've made a big deal out of the fact that one thread of control keeps things simple and easy to program, and that is true. But, as you're going to see, Web Workers have been carefully crafted to make sure things stay simple, easy and safe for the programmer. We'll see how in just a moment...

JavaScript Exposed (Again)

This week's interview: Where JavaScript spends his time.

Head First: Welcome back JavaScript, great to have you.

JavaScript: Glad to be here, as long as we stick to my schedule, lots to do.

Head First: That is actually where I thought we might focus our time today. You're a super successful guy, you have so much going on—how do you get it all done?

JavaScript: Well, I have a philosophy: I do one thing at a time, and I do it really well.

Head First: How do you do only one thing at a time? To us it looks like you're retrieving data, displaying pages, interacting with the user, managing timers and alerts, and on and on...

JavaScript: Yes, I do all that, but whatever I'm doing, I do only that. So if I'm dealing with the user, that's all I do until I'm done with that.

Head First: How can that be true? What if a timer goes off, or network data arrives, or whatever, don't you stop and do that?

JavaScript: When an event occurs, like the ones you've mentioned, that event is added to a queue. I don't even look at it until I've finished whatever I'm working on. That way I do everything correctly and safely and efficiently.

Head First: Are you ever late getting to one of those tasks on the queue?

JavaScript: Oh it happens. Luckily I'm the technology behind browser web pages, so how bad can it be if I get a little behind? You should talk to the guys that have to run code for spacecraft thrusters or nuclear power plant controllers, those guys have to live by different rules—that's why they make the big bucks.

Head First: I've always wondered what's going on when I get the "Slow script, do you want to continue" dialog on my browser. Is that you taking a break?

JavaScript: Taking a break! Hah. That's when someone has structured their page such that I've got so much work to do, I can't do it all! If you write a bit of JavaScript that hogs all my time, then your interaction with your user is going to suffer. I can only do so much.

Head First: Sounds like you need some help.

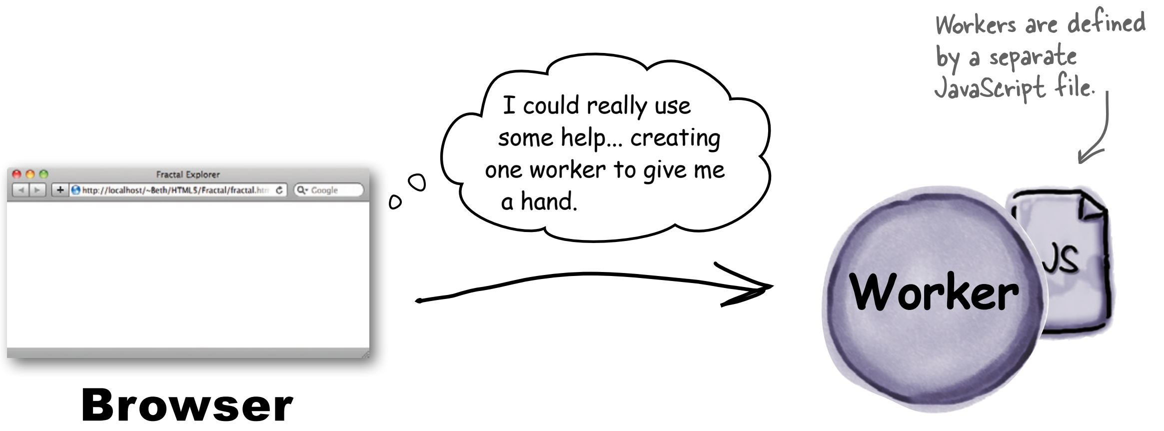
JavaScript: Well thanks to HTML5, I have help now because that's where Web Workers come in. If you really need to write compute-intensive code, use Web Workers to offload some of the work—that way I can keep my focus, and workers can do some of the heavy lifting for me (without getting in my way).

Head First: Interesting, we'll look into that. Now, next question... Oh, wait, he's gone, looks like he's off to his next task. Serious guy, huh?

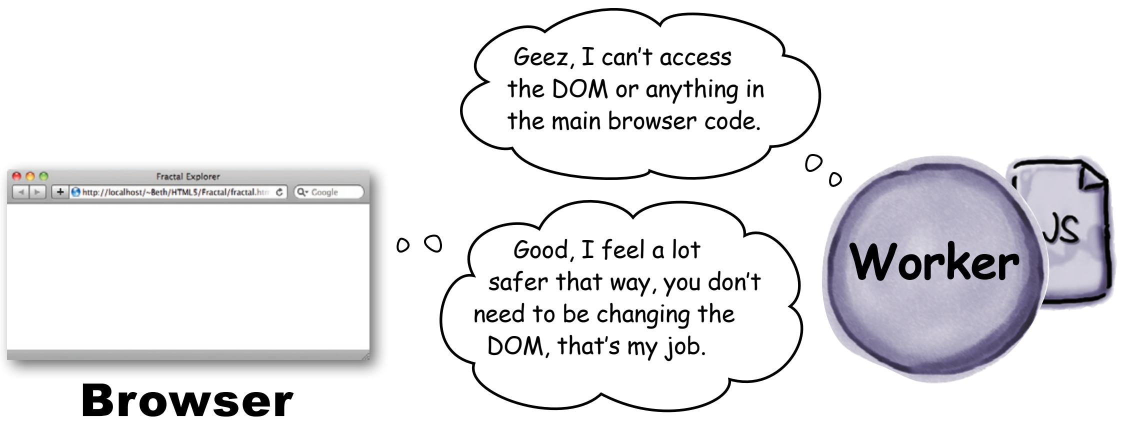
How Web Workers work

Let's take a look at a day in the life of a Web Worker: how workers are created, how they know what to do, and how they get results back to your main browser code.

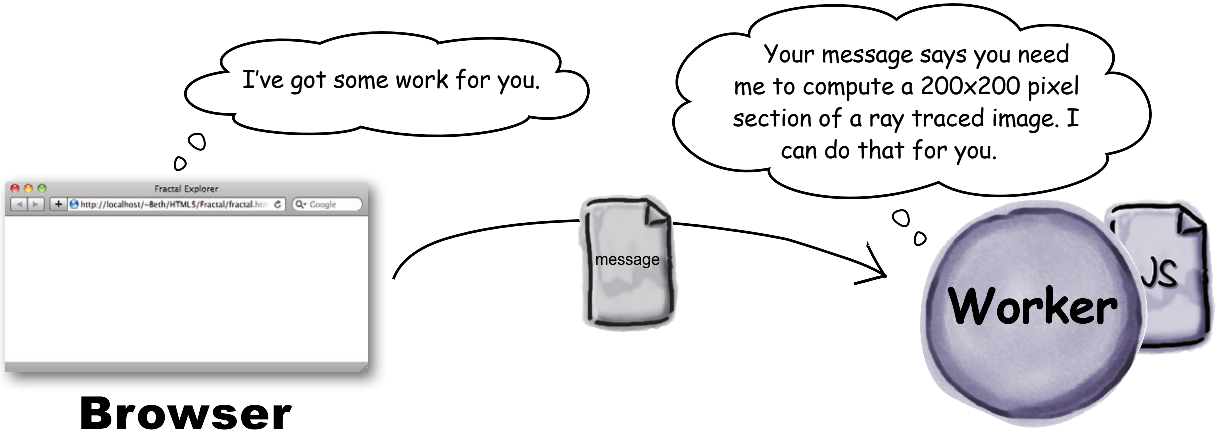
To use Web Workers, the browser first has to create one or more workers to help compute tasks. Each worker is defined with its own JavaScript file that contains all the code (or references to code) it needs to do its job.



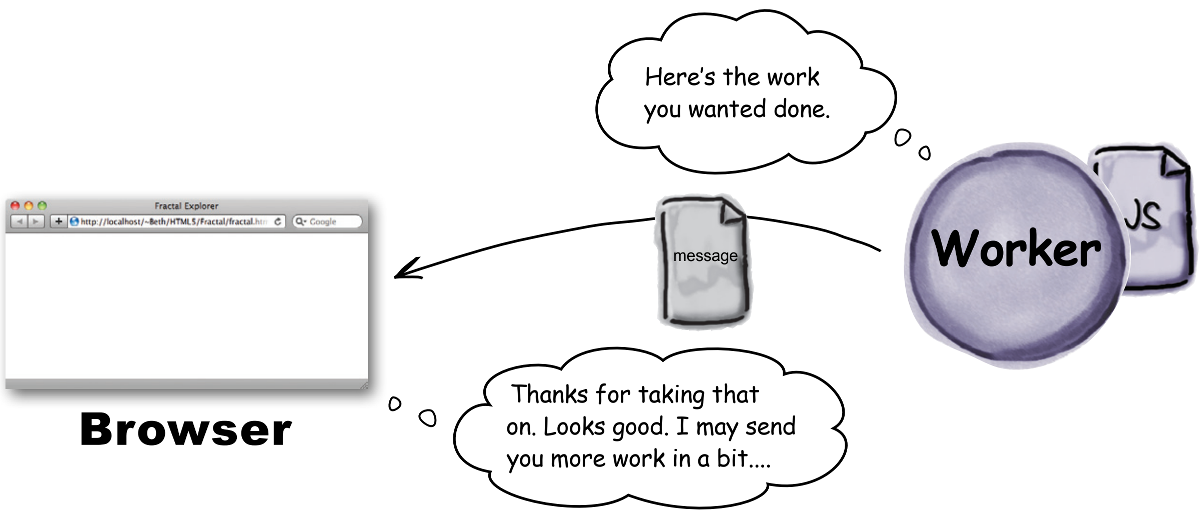
Now, workers live in a very restricted world; they don't have access to many of the runtime objects your main browser code does, like the DOM or any of the variables or functions in your main code.



To get a worker to start working, the browser typically sends it a message. The worker code receives the message, takes a look at it to see if there are any special instructions, and starts working.



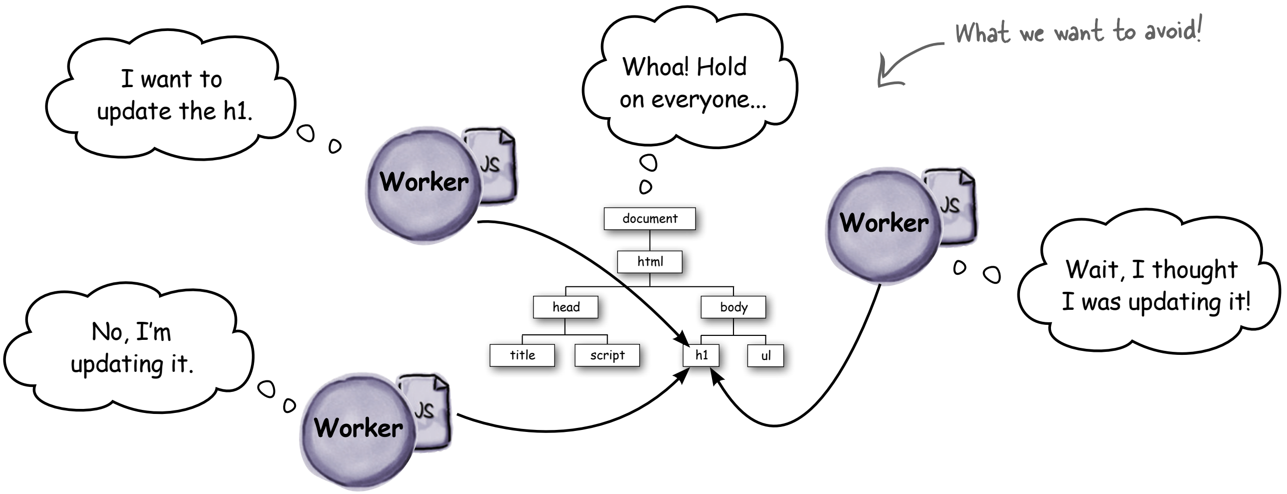
When the worker completes its work, it then sends a message back, with the final results of what it's been working on. The main browser code then takes these results and incorporates them into the page in some way.



Why not allow workers to access the DOM? I mean this seems like a lot of trouble to pass messages back and forth when all of these workers are running in the same browser.

To keep things efficient.

One reason the DOM and JavaScript have been so successful is that we're able to highly optimize DOM operations because we have only one thread with access to the DOM. If we let multiple threads of computation concurrently change the DOM, then we'll seriously impact its performance (and browser implementors would have to go to great effort to make sure making changes to the DOM is safe). The truth is, allowing a bunch of changes to the DOM at the same time can easily lead to situations where the DOM is in an inconsistent state, which would be bad. Very bad.



Sharpen your pencil

Take a look at all the potential uses for workers below. Which ones might improve the design and the performance of an app?

- [ ] Caching data for use in your pages.
- [ ] Processing large amounts of data in arrays or large JSON responses from web services.
- [ ] Managing a database connection along with adding and removing records for the main page.
- [ ] Automated race track betting agent.
- [ ] Analyzing video.
- [ ] ....
- [ ] ....
- [ ] ....
- [ ] Spell checking the page as the user types.
- [ ] Polling web services and alerting the main page when something happens.
- [ ] Image processing of data in a canvas.
- [ ] Code syntax or other language highlighting.
- [ ] Pre-fetching data based on what your user is doing.
- [ ] Managing advertising for your page.
- [ ] ....
- [ ] ....
- [ ] ....

Your ideas here!

All answers are good uses, although you could debate a couple: spell checking & syntax checking may be better done in the main page code; race track betting may be better not done at all. ;-)

Watch it!

Google's Chrome browser has some extra security restrictions that will prevent you from running Web Workers directly from a file. If you try, your page won't run and you'll get no indication of why (including no error messages telling you what's wrong!).

So, for these examples, we recommend either using a different browser, or running your own server and running the examples from http://localhost. Or you can upload them to a hosted server if you have access to one.

You can also use the Chrome runtime switch --allow-file-access-from-files, but we don't recommend this switch beyond just testing your code.

Watch it!

Almost all the modern browsers support Web Workers, but there is one exception: Internet Explorer 9. The good news is that for IE10 and later, you can count on Web Workers, but with IE9 and all the IE versions before it, you'll have to supply an alternative experience.

But rather than worrying about IE specifically, here's how you can easily check to see if any browser supports Web Workers:

If workers are supported, the property Worker will be defined in the global scope, window.

And if Worker isn't defined, then we've got no support in the browser.

```
if (window["Worker"]) {
    var status = document.getElementById("status");
    status.innerHTML = "Bummer, no Web Workers";
}
```

You'll want to handle that condition in a way that is appropriate for your app. Here, we're just letting the user know by putting a message in an element with id="status".

Your first Web Worker...

Let's get down to the business of creating a worker to see how this all works. To do that we need a page to put everything in. We'll go with the simplest HTML5 markup we can get away with; type this into `pingpong.html`:

Web Worker

```
<!doctype html>
<html lang="en">
   <head>
      <title>Ping Pong</title>
      <meta charset="utf-8">
      <script src="manager.js"></script>
   </head>
<body>
   <p id="output"></p>
</body>
</html>
```

This JavaScript code is going to create and manage all the workers.

And we'll be putting some output from the worker here.

How to create a Web Worker

Before we start implementing `manager.js`, let's look at how you actually create a Web Worker:

To create a new worker, we create a new Worker object...

```
var worker = new Worker("worker.js");
```

And we're assigning the new worker to a JavaScript variable named worker.

... the "worker.js" JavaScript file contains the code for the worker.

So that's how you create one worker, but of course you don't have to stop there; you can create as many workers as you like:

```
var worker2 = new Worker("worker.js");
var worker3 = new Worker("worker.js");

var another_worker = new Worker("another_worker.js");
```

We can easily create two more workers that make use of the same code as our first worker.

We'll see how to use multiple workers together in a bit...

Or we can create other workers based a different JavaScript file.

Writing Manager.js

Now that you know how to create a worker (and how easy it is), let's start working on our `manager.js` code. We'll keep this code simple and create just one worker for now. Create a file named `manager.js` and add this code:

```
window.onload = function() {
    var worker = new Worker("worker.js");
}
```

We'll wait for the page to fully load.

And then create a new worker.

That's a great start, but now we want to get the worker to actually do some work. As we've already discussed, one way we tell a worker to do some work is by sending it a message. To send a message we use the worker object's `postMessage` method. Here's how you use it:

```
window.onload = function() {
    var worker = new Worker("worker.js");

    worker.postMessage("ping");
}
```

And we're using the worker's postMessage method to send it a message. Our message is the simple string "ping".

The postMessage method is defined for you in the Web Worker API.

Want to send more complex messages? Here's how...

postMessage Up Close

You can send more than just strings in `postMessage`. Let's look at everything you can send in a message:

```
worker.postMessage("ping");
worker.postMessage([1, 2, 3, 5, 11]);
worker.postMessage({"message": "ping", "count": 5});
```

You can send a string...

... an array...

... or even a JSON object.

You **can't** send functions:

```
worker.postMessage(updateTheDOM);
```

You can't send a function... it might contain a reference to the DOM allowing the worker to change the DOM!

Receiving messages from the worker

We're not quite done with our `manager.js` code yet—we still need to be able to receive a message from the worker if we're going to make use of all its hard work. To receive a worker's message we need to define a handler for the worker's `onmessage` property so that anytime a message comes in from that worker, our handler will be called (and handed the message). Here's how we do that:

```
window.onload = function() {
    var worker = new Worker("worker.js");

    worker.postMessage("ping");

    worker.onmessage = function (event) {
        var message = "Worker says " + event.data;
        document.getElementById("output").innerHTML = message;
    };
}
```

Here we're defining a function that will get called whenever we receive a message from this worker. The message from the worker is wrapped in an event object.

The event object passed to our handler has a data property that contains the message data (what we're after) that the worker posted.

When we get a message from the worker we'll stuff it in a <p> element in the HTML page.

onMessage Up Close

Let's take a quick look at the message our `onmessage` handler is receiving from the worker. As we've said, this message is wrapped in an `Event` object, which has two properties we're interested in: `data` and `target`:

This is the object that is sent from the worker to the code in your page when the worker posts a message.

```
worker.onmessage = function (event) {
    var message = event.data;
    var worker = event.target;
};
```

The data property contains the message the worker sent (e.g., a string, like "pong").

And the target is a reference to the worker that sent the message. This can come in handy if you need to know which worker it's from. We'll be using this later in the chapter.

Now let's write the worker

To get started on the worker, the first thing we need to do is to make sure the worker can receive messages that are sent from `manager.js`—that's how the worker gets its work orders. For that we're also going to make use of another `onmessage` handler, the one in the worker itself. Every worker is ready to receive messages, you just need to give the worker a handler to process them. Here's how we do that (go ahead and create a file `worker.js` and add this code):

```
onmessage = pingPong;
```

We're assigning the onmessage property in the worker to the pingPong function.

We're going to write the function pingPong to handle any messages that come in.

Writing the worker's message handler

Let's write the worker's message handler, `pingPong`, and we're going to start simple. Here's what it's going to do (you might have already guessed from the name `pingPong`): the worker's going to check any message it gets to make sure it contains the string "ping", and if it does, we're going to send a message back that says "pong". So, in effect, the work of the worker is just to get a "ping" and to answer with a "pong"—we're not going to do any heavy computation here, we're just going to make sure the manager and worker are communicating. Oh, and if the message doesn't say "ping", we're just going to ignore it.

So the function `pingPong` takes a message and responds with "pong". Go ahead and add this code to `worker.js`:

When the worker receives a message from the main code, the pingPong function will be called, and the message will be passed in.

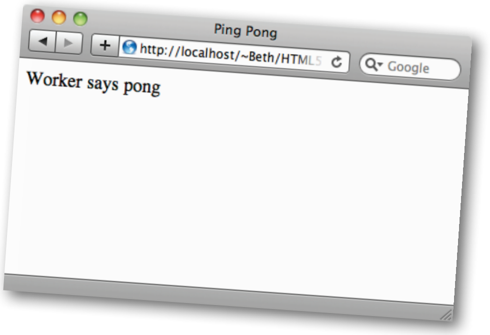
```
onmessage = pingPong;
function pingPong(event) {
    if (event.data == "ping") {
        postMessage("pong");
    }
}
```

And if the message contains a string that says "ping", we'll send back a message that says "pong". The worker's message goes back to the code that created the worker.

Notice the worker uses postMessage to send messages, too.

Serving up a test drive 

Make sure you've got `pingpong.html`, `manager.js` and `worker.js` typed in and saved. Now keep those files open so you can review them and let's think through how this works. First, `manager.js` creates a new worker, assigns a message handler to it, and then sends the worker a "ping" message. The worker, in turn, makes sure `pingPong` is set up as its message handler, and then it waits. At some point, the worker receives a message from the manager, and when it does it checks to see that it contains "ping", which it does, and then the worker does ~~a lot of~~ very little hard work and sends a "pong" message back.



At this point the main browser code receives a message from the worker, which it hands to the message handler. The handler then simply prepends "Worker says " to the front of the message, and displays it.

Now, our calculations here say the page should display "Worker says pong"...okay okay, we know, you can't take the suspense any more... go ahead and load the page already!

BE the Browser

It's time to pretend you're the browser evaluating JavaScript. For each bit of code below, act like you're the browser and write its output in the lines provided. You can assume this code is using the same worker.js we just wrote:

You can check the solutions at the end of the chapter.

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
    }
    for (var i = 0; i < 5; i++) {
        worker.postMessage("ping");
    }
}
```

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.ommessage = function(event) {
        alert("Worker says " + event.data);
    }
    for(var i = 5; i > 0; i--) {
        worker.postMessage("pong");
    }
}
```

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
        worker.postMessage("ping");
    }
    worker.postMessage("ping");
}
```

Careful if you try these; you might have to kill your browser to escape...

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
    }

    setInterval(pinger, 1000);

    function pinger() {
        worker.postMessage("ping");
    }
}
```

Sharpen your pencil

While workers typically get their work orders through a message, they don't have to. Check out this nice, compact way to get work done with workers and HTML. When you know what it does, describe it below. You can check your solution with ours at the end of the chapter.

quote.html

```html
<!doctype html>
<html lang="en">
  <head>
    <title>Quote</title>
    <meta charset="utf-8">
  </head>
<body>
  <p id="quote"></p>
  <script>
      var worker = new Worker("quote.js");
      worker.onmessage = function(event) {
          document.getElementById("quote").innerHTML = event.data;
      }
  </script>
</body>
</html>
```

quote.js

```javascript
var quotes = ["I hope life isn't a joke, because I don't get it.",
              "There is a light at the end of every tunnel... just pray it's not a train!",
              "Do you believe in love at first sight or should I walk by again?"];
var index = Math.floor(Math.random() * quotes.length);
postMessage(quotes[index]);
```

Your description here:

Try typing in the code and running it!

Exercise

Let's add a couple of workers to our pingPong game. Your job is to fill in the blanks to complete the code so we have three pings sent to the workers, and three pongs back from the workers.

```
window.onload = function() {
    var numWorkers = 3;
    var workers = [];
    for (var i = 0; i < ..........................; i++) {
        var worker = new ........................("worker.js");
        worker. ..................... = function(event) {

            alert(event.target + " says "
                                  + event. ................);
        };
        workers.push(worker);
    }
    for (var i = 0; i < ..........................; i++) {
        workers[i]. ........................("ping");
    }
}
```

We're creating three workers, and storing them in an array, workers.

Write your code in the blanks.

Here, we're adding the new worker to the workers array.

there are no Dumb Questions

Q: Can I just pass a function instead of a JavaScript file when I create the worker? That would seem easier and more consistent with how JavaScript usually does things.

A: No, you can't. Here's why: as you know, one of the requirements of a worker is that it not have access to the DOM (or to any state of the main browser thread for that matter). If you could pass a function to the Worker constructor, then your function could also contain reference to the DOM or other parts of your main JavaScript code, which would violate that requirement. So, the designers of Web Workers chose instead to have you just pass a JavaScript URL to avoid that issue.

Q: When I send a worker an object in a message, does it become a shared object between my main page and the worker?

A: No, when you send an object the worker gets a copy of it. Any changes the worker makes will not affect the object in your main page. The worker is executing in a different environment than your main page, so you have no access to objects there. The same is true of objects the worker sends you: you get a copy of them.

Q: Can workers access localStorage or make XMLHttpRequests?

A: Yes, workers can access localStorage and make XMLHttpRequests.

Exercise Solution

Let's add a couple of workers to our pingPong game. Your job was to fill in the blanks to complete the code so we have three pings sent to the workers and three pongs back from the workers. Here's our solution.

We use numWorkers to iterate three times and create three workers (feel free to change this variable to add more!)

```
window.onload = function() {
    var numWorkers = 3;
    var workers = [];
    for (var i = 0; i < numWorkers; i++) {
        var worker = new Worker("worker.js");
        worker.onmessage = function(event) {

            alert(event.target + " says "
                               + event.data);
        };
        workers.push(worker);
    }
    for (var i = 0; i < workers.length; i++) {
        workers[i].postMessage("ping");
    }
}
```

We set up the message handler in our main page code by using the onmessage property of the worker.

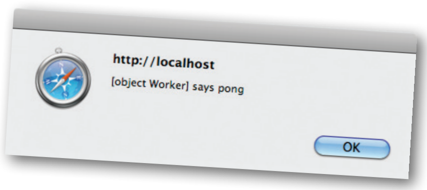
We use the data property to get the contents of the message.

You could also use numWorkers here if you like.

We ping the worker with postMessage.

Notice that no changes are needed to the worker code. Each worker is happy to do its thing independently.

You'll see this alert 3 times.



Take a look at importScripts.

Web Workers have a global function named `importScripts` that you can use to import one or more JavaScript files into your worker. To use `importScripts` just give it a comma separated list of files or URLs you'd like to import, like this:

```
importScripts("http://bigscience.org/nuclear.js",
              "http://nasa.gov/rocket.js",
              "mylibs/atomsmasher.js");
```

Place zero or more comma-separated JavaScript URLs in importScripts.

Then when `importScripts` is invoked, each JavaScript URL is retrieved and evaluated in order.

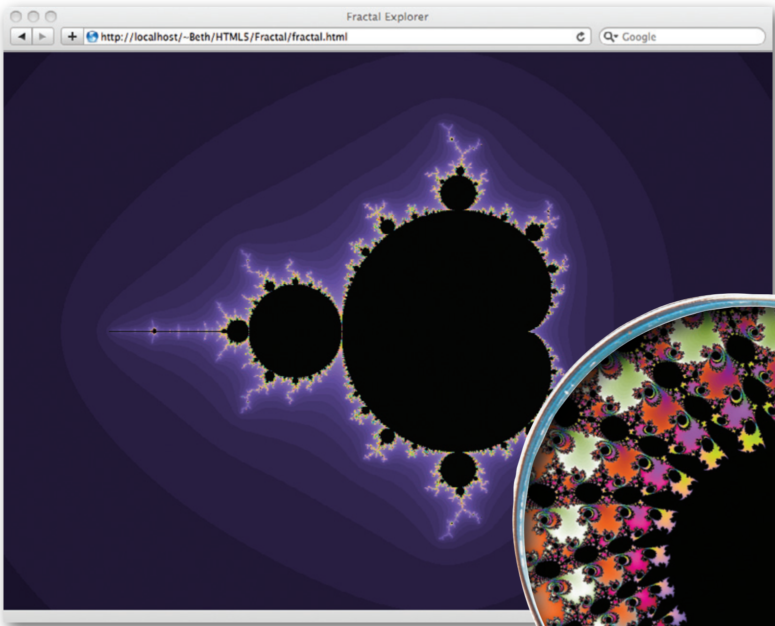
Notice that `importScripts` is a full-fledged function, so (unlike `import` statements in a lot of languages) you can make runtime decisions about importing, like this:

```
if (taskType == "songdetection") {
   importScripts("audio.js");
}
```

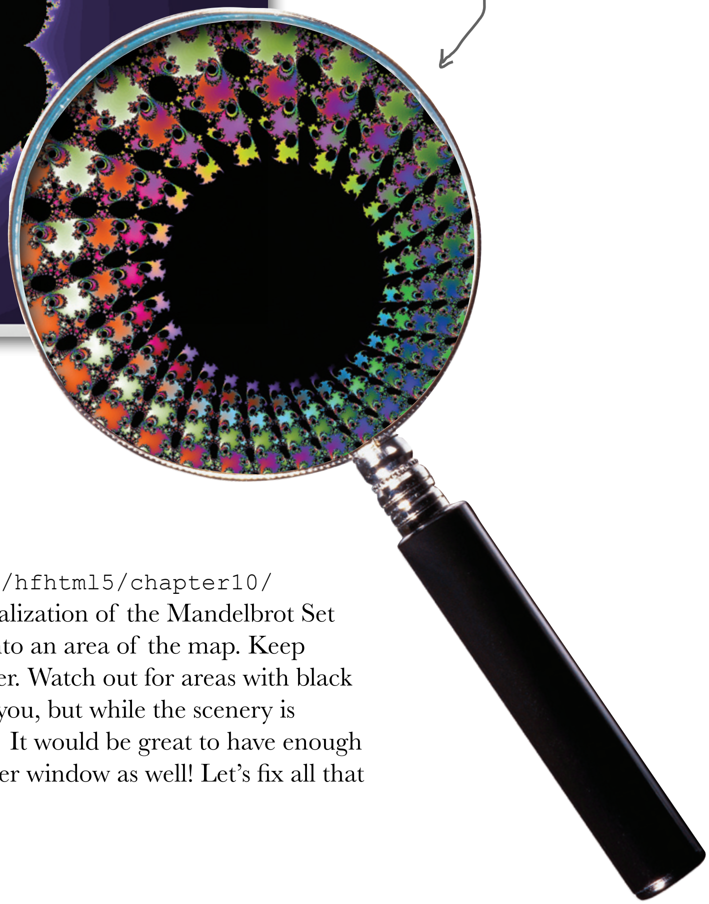
Because importScripts is a function, you can import code as the task demands.

Virtual Land Grab

Explorers of the Mandelbrot Set have already grabbed areas of the virtual countryside and given them names like the lovely "Seahorse Valley," "Rainbow Islands," and the dreaded "Black Hole." And given the value of physical real estate these days, the only play left seems to be in the virtual spaces. So, we're going to build an explorer for the Mandelbrot Set to get in on the action. Actually, we have to confess, we already have built it, but it's slow—navigating around in the entire Mandelbrot Set could take a very long time —so we're hoping together we can speed it up, and we have a hunch Web Workers may be the answer.

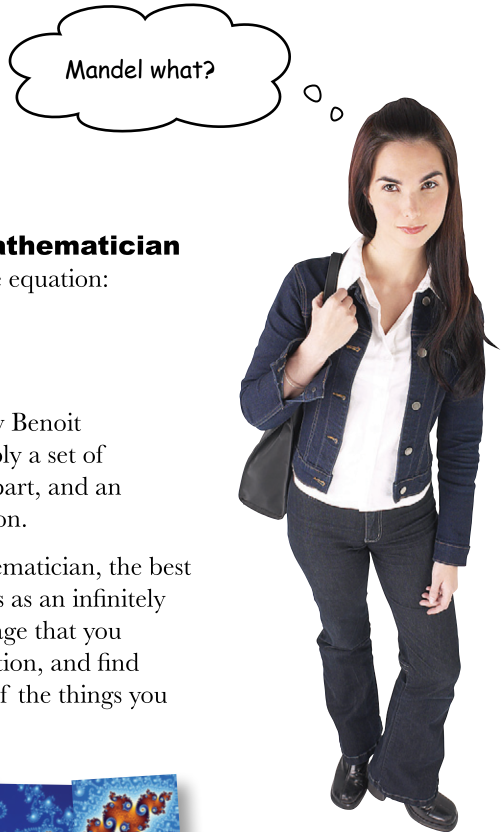


Like some beach front property right on the edge of the Azure Vortex?



Take a look around

Go ahead and fire up `http://wickedlysmart.com/hfhtml5/chapter10/singlethread/fractal.html` and you'll see a visualization of the Mandelbrot Set in the distance. Just click anywhere and you'll zoom into an area of the map. Keep clicking to explore different areas, or reload to start over. Watch out for areas with black holes, they tend to suck you in. We don't know about you, but while the scenery is beautiful, our viewer could be a little faster... ya think? It would be great to have enough performance to maximize the view to the entire browser window as well! Let's fix all that by adding Web Workers to the Fractal Explorer.

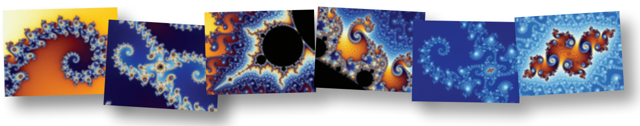


Well, if you happen to be a mathematician
then you know the Mandelbrot Set is the equation:

$$z_n+1 = z_n^2 + c$$

and that it was discovered and studied by Benoit Mandelbrot. You also know that it's simply a set of complex numbers (numbers with a real part, and an imaginary part) generated by this equation.

If, on the other hand, you're not a mathematician, the best way to think about the Mandelbrot Set is as an infinitely complex fractal image—meaning an image that you can zoom into, to any level of magnification, and find interesting structures. Just look at some of the things you can find by navigating into the set:



And why are we so interested in it? Well, the set has a few interesting properties. First, it's generated by a very simple equation (the one above) that can be expressed in just a few lines of code; second, generating the Mandelbrot Set takes a fair number of computing cycles, which makes it a great example for using Web Workers. And finally, hey, it's cool and a trip to work with, and what a great app to end the book with, don't you think?

RIP Benoit Mandelbrot, who passed during the writing of this book. We were lucky to have known you.

How to compute a Mandelbrot Set

Let's take a look at how you'd typically structure your code to compute a Mandelbrot Set before we get workers involved. We don't want to focus much on the nitty-gritty of computing Mandelbrot pixel values; we've already got all that code taken care of, and we're going to give it to you in a sec. For now, we just want you to get sense of the big picture view of how to compute the set:

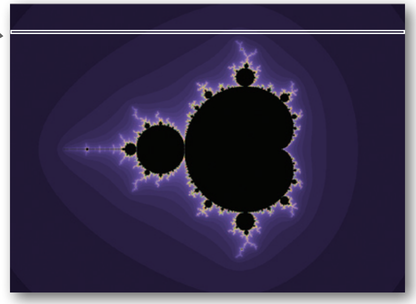
Note, our aim here isn't to teach you to be a numerical analyst (who can code equations with complex numbers); it's to adapt a compute intensive application to use Web Workers. If you are interested in the numerical aspects of the Mandelbrot Set, Wikipedia is a great place to start.

To compute the Mandelbrot Set we loop over each row of the image.

```
for (i = 0; i < numberOfRows; i++) {
    var row = computeRow(i);
    drawRow(row);
}
```

And for each row we compute the pixels for that row.

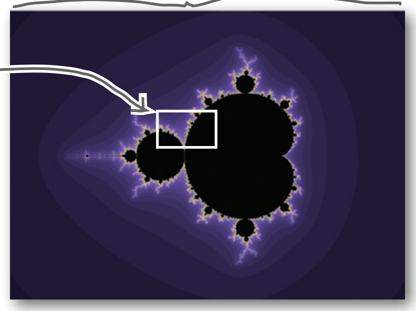
And then we draw each row on the screen. You can probably see the row-by-row display when you run the test code in your browser.



Now this code is just meant to be simple pseudo-code—when it comes to writing the code for real, there are a few more details we need to get into: for instance, to compute a row we need to know the width of the row, the zoom factor, the numerical resolution to which we want to compute it, and a few other small details. We can capture all those details in a task object like this:

width

The zoom factor



Level of precision to compute

```
for (i = 0; i < numberOfRows; i++) {
    var taskForRow = createTaskForRow(i);
    var row = computeRow(taskForRow);
    drawRow(row);
}
```

The taskForRow object holds all the data needed to compute a row.

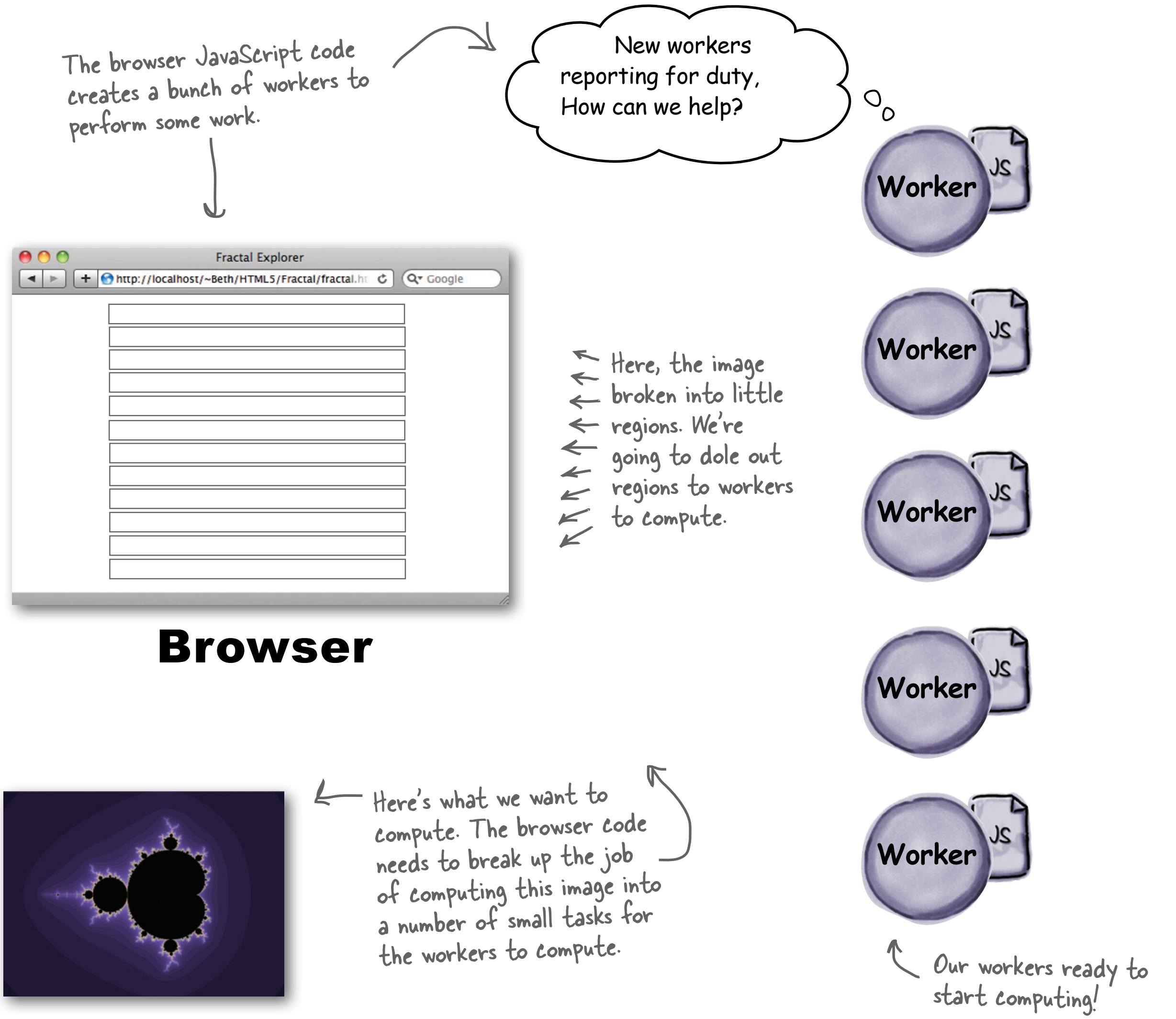
And we pass taskForRow into computeRow, which returns the computed row.

Now the trick is going to be taking this and reworking it to divide up the computation among a number of workers, and then adding the code that handles giving tasks to workers, and handles dealing with the results when the workers complete the tasks.

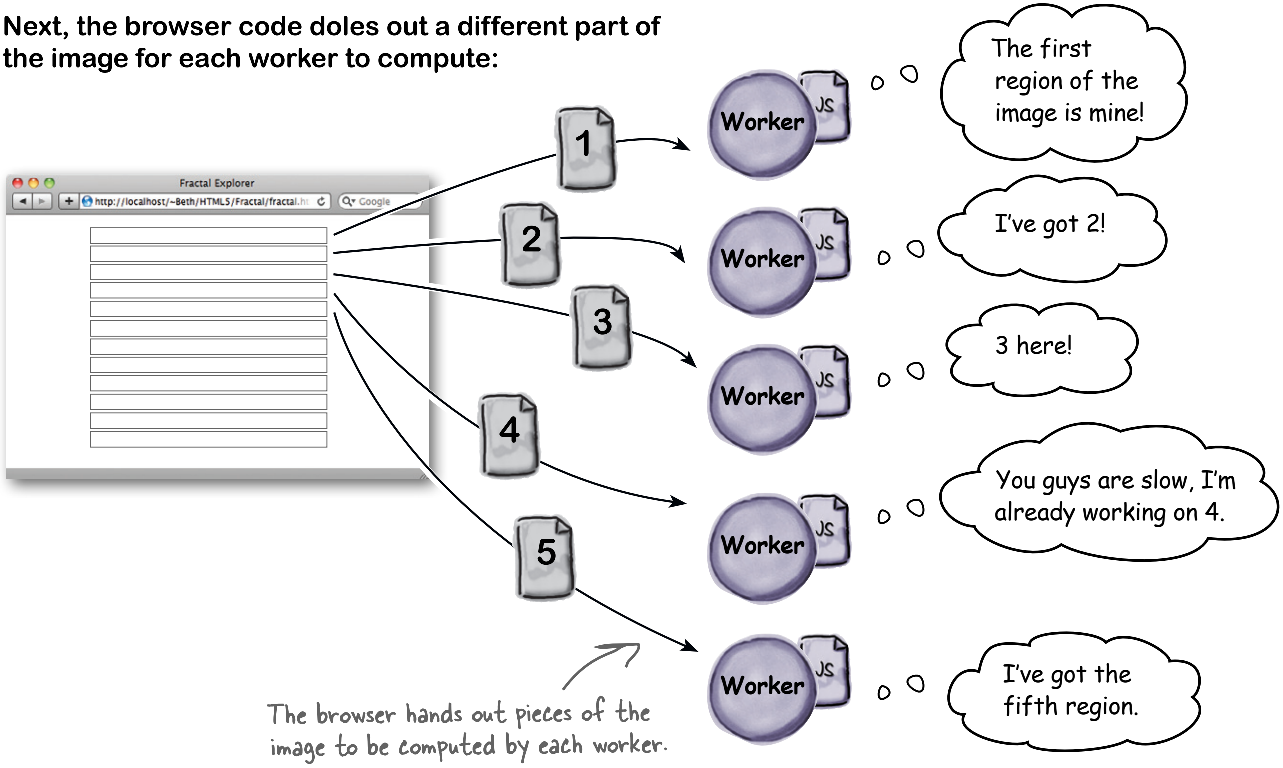
How to use multiple workers

You already know how to create new workers, but how do you use them to do something a little more complicated, like computing the rows of the Mandelbrot Set? Or applying a Photoshop-like effect over an image? Or ray tracing a movie scene? In all these cases, we can break up the job into small tasks that each worker can work on independently. For now, let's stick with computing the Mandelbrot Set (but the pattern we're going to use can be applied to any of these examples).

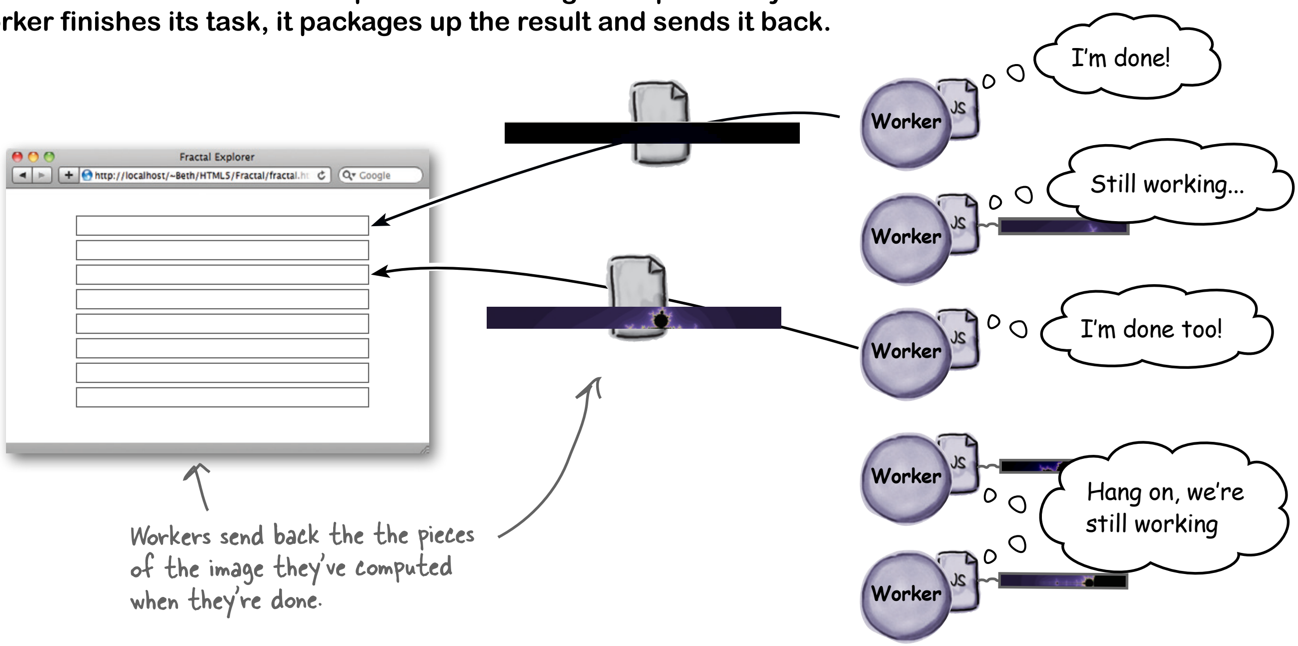
To get started, the browser first creates a bunch of workers to help (but not too many—workers can be expensive if we create too many of them—more on this later). We'll use just five workers for this example:



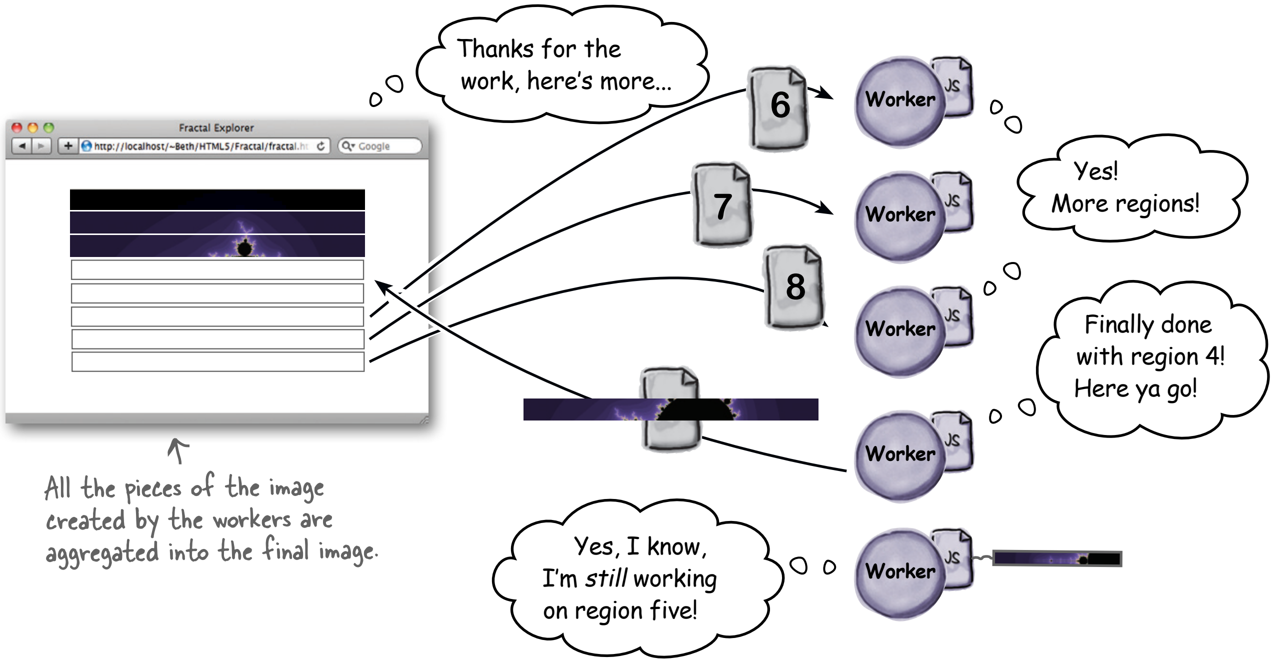
Next, the browser code doles out a different part of the image for each worker to compute:



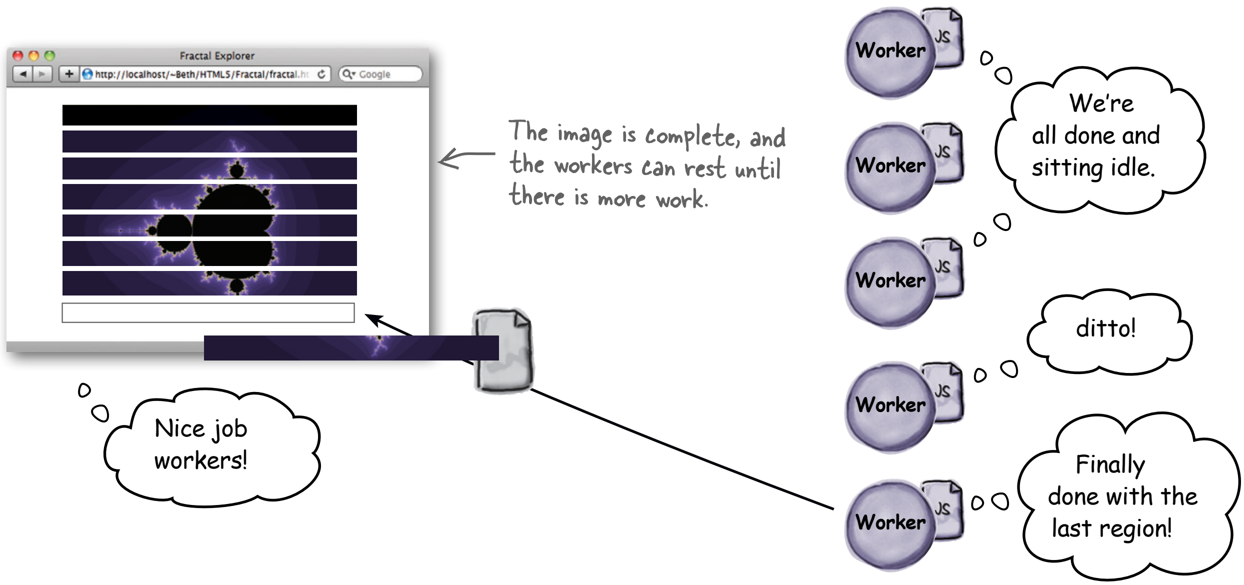
Each worker works on its own piece of the image independently. As a worker finishes its task, it packages up the result and sends it back.

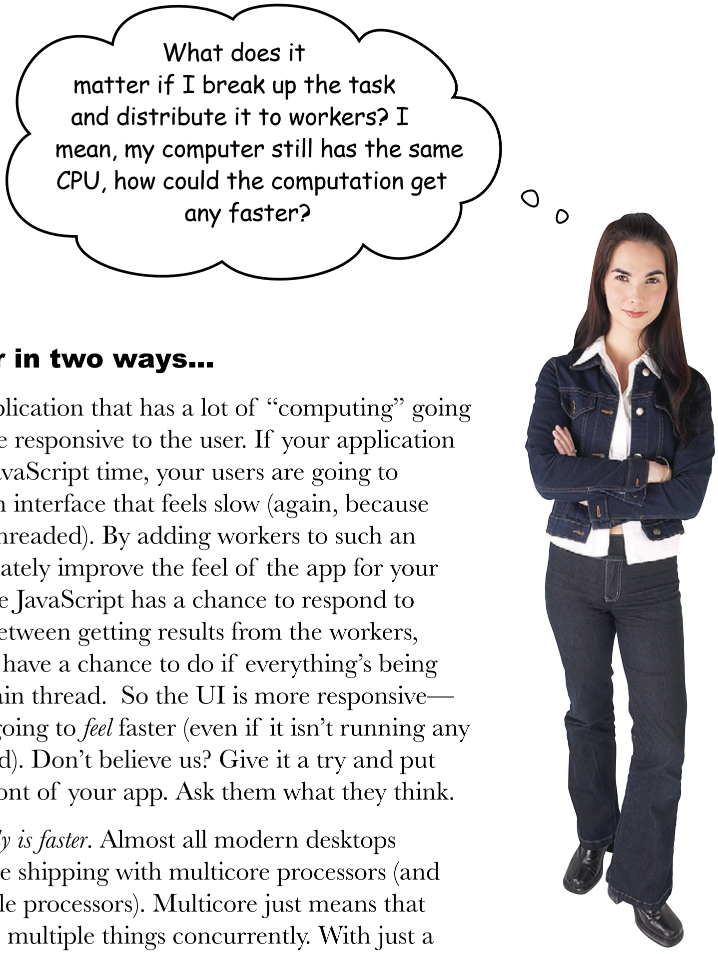


As pieces of the image come back from the workers they are aggregated into the image in the browser, and if there are more pieces to compute, new tasks are handed out to the workers that are idle.



With the last piece of the image computed, the image is complete and the workers sit idle, until the user clicks to zoom in, and then it all starts again...

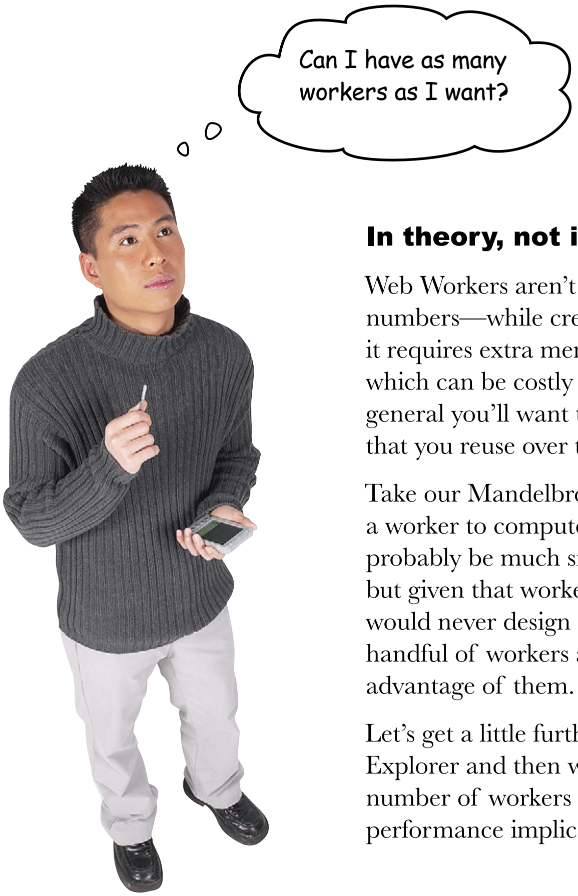




It can be faster in two ways...

First consider an application that has a lot of "computing" going on that also has to be responsive to the user. If your application is hogging a lot of JavaScript time, your users are going to experience a sluggish interface that feels slow (again, because JavaScript is single-threaded). By adding workers to such an app you can immediately improve the feel of the app for your users. Why? Because JavaScript has a chance to respond to user interaction in between getting results from the workers, something it doesn't have a chance to do if everything's being computed on the main thread. So the UI is more responsive—and your app's just going to *feel* faster (even if it isn't running any faster under the hood). Don't believe us? Give it a try and put some real users in front of your app. Ask them what they think.

The second way *really is faster*. Almost all modern desktops and devices today are shipping with multicore processors (and perhaps even multiple processors). Multicore just means that the processor can do multiple things concurrently. With just a single thread of control, JavaScript in the browser doesn't make use of your extra cores or your extra processors, they're just wasted. However, if you use Web Workers, the workers can take advantage of running on your different cores and you'll see a real speedup in your app because you've got more processor power being thrown at it. If you've got a multicore machine, just wait, you're going to see the difference soon.



In theory, not in practice.

Web Workers aren't intended to be used in large numbers—while creating a worker looks simple in code, it requires extra memory and an operating system thread, which can be costly in start-up time and resources. So, in general you'll want to create a limited number of workers that you reuse over time.

Take our Mandelbrot example: in theory you could assign a worker to compute every single pixel, which would probably be much simpler from a code design perspective, but given that workers are heavy-weight resources, we would never design our app that way. Instead, we'll use a handful of workers and structure our computation to take advantage of them.

Let's get a little further into the design of the Fractal Explorer and then we'll come back and play with the number of workers we're using to understand the performance implications.

BRAIN POWER

You've certainly got a lot of background now on building Web Worker apps, how to create and use workers, a bit about how you can solve big computations by breaking them down into small tasks that can be computed by your workers, and you even know a little about how Mandelbrot sets are computed. Try to put it all together and think through how you'd take the pseudo-code below and rewrite it to use workers. You might first assume you have as many workers as you need (say a worker for every single row), and then add the constraint that you have a limited number of workers (fewer workers than the number of rows):

```
for (i = 0; i < numberOfRows; i++) {
    var taskForRow = createTaskForRow(i);
    var row = computeRow(taskForRow);
    drawRow(row);
}
```

Here's our pseudo-code now, what do you need to do to add Web Workers?

Your notes go here:

Let's build the Fractal Explorer app

- [ ] Create HTML
- [ ] Ready Bake Code
- [ ] Create workers
- [ ] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

Here's what we need to do:

- [ ] Set up our HTML page to hold the Mandelbrot App.
- [ ] Get all the Ready Bake Code entered (or downloaded).
- [ ] Create some workers and get them set up to compute.
- [ ] Start the workers on their tasks.
- [ ] Implement the worker code.
- [ ] Process the worker results as the workers complete their tasks.
- [ ] Handle click and resize events in the user interface.

Creating the Fractal Viewer HTML Markup

First we need to set up an HTML page to hold our app. You'll want to create an HTML file named `fractal.html` and add the following markup. Let's check it out:

```
<!doctype html>
<html lang="en">
   <head>
      <title>Fractal Explorer</title>
      <meta charset="utf-8">
      <link rel="stylesheet" href="fractal.css">
      <script src="mandellib.js"></script>
      <script src="mandel.js"></script>
   </head>
   <body>
      <canvas id="fractal" width="800" height="600"></canvas>
   </body>
</html>
```

As usual, a standard HTML5 file.

Here's all the Ready Bake Code we've got for you, this contains all the numerical code as well as some code for handling graphics.

And here's the JavaScript code we're going to be writing...

If you're wondering where the worker code is going to go, remember, we don't link directly to a worker JavaScript file, we reference that file when we create the worker in code.

Look! Our friend the <canvas> is back!!

And the <body> has a canvas element. We set it to an initial size of 800 x 600 pixels, but we'll see how to resize it to the width and height of the window using JavaScript. After all we want as large a Mandelbrot as we can get!

This code goes in fractal.html

Ready Bake Code

Reminder: you can download all the code from http://wickedlysmart.com/hfhtml5

We have to tell you we were planning on an entire chapter on the wonders of computing the Mandelbrot Set... we planned to explain it to you in detail, including a history of Benoit Mandelbrot, how he discovered it, all its amazing properties, pixel optimizations, color maps, and so on, but then we got the call from our editor—you know, THE CALL. I guess we were running a bit late on this book, so our apologies, but we're going to have to give you some **Ready Bake Code** code to do the low-level computation of the Mandelbrot graphics. Here's the good side though: we can focus on how to use Web Workers without spending the next couple of days on math and graphics. That said, we encourage you to explore those topics on your own!

Anyway, first we've got the code used to manage tasks and draw the rows for the fractal images. Start by typing this code into a file called "mandellib.js":

```
var canvas;
var ctx;
```

Notice our canvas and context are here.

```
var i_max = 1.5;
var i_min = -1.5;
var r_min = -2.5;
var r_max = 1.5;
```

These are the global variables the Mandelbrot graphics code uses to compute the set and display it.

```
var max_iter = 1024;
var escape = 1025;
var palette = [];
```

This function packages up all the data needed for the worker to compute a row of pixels, into an object. You'll see later how we pass this object to the worker to use.

```
function createTask(row) {
    var task = {
        row: row,
        width: rowData.width,
        generation: generation,
        r_min: r_min,
        r_max: r_max,
        i: i_max + (i_min - i_max) * row / canvas.height,
        max_iter: max_iter,
        escape: escape
    };
    return task;
}
```

This code goes in mandellib.js.

Ready Bake Code Continued...

- [x] Create HTML
- [ ] Ready Bake Code
- [ ] Create workers
- [ ] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

```
function makePalette() {
    function wrap(x) {
       x = ((x + 256) & 0x1ff) - 256;
       if (x < 0) x = -x;
        return x;
    }
    for (i = 0; i <= this.max_iter; i++) {
        palette.push([wrap(7*i), wrap(5*i), wrap(11*i)]);
    }
}
```

makePalette maps a large set of numbers into an array of rgb colors. We'll use this palette in drawRow (below) to convert the value we get back from a worker to a color for the graphic display of the set (the fractal image).

drawRow takes the results from the worker and draws them into the canvas.

```
function drawRow(workerResults) {
    var values = workerResults.values;
    var pixelData = rowData.data;
    for (var i = 0; i < rowData.width; i++) {
        var red = i * 4;
        var green = i * 4 + 1;
        var blue = i * 4 + 2;
        var alpha = i * 4 + 3;
        pixelData[alpha] = 255; // set alpha to opaque
        if (values[i] < 0) {
            pixelData[red] = pixelData[green] = pixelData[blue] = 0;
        } else {
            var color = this.palette[values[i]];
            pixelData[red] = color[0];
            pixelData[green] = color[1];
            pixelData[blue] = color[2];
        }
    }
    ctx.putImageData(this.rowData, 0, workerResults.row);
}
```

It uses this rowData variable to do it; rowData is a one-row ImageData object that holds the actual pixels for that row of the canvas.

Here's where we use the palette to map the result from the worker (just a number) to a color.

This code should be familiar; it's similar to what we did in Chapter 8 with video and canvas.

And here's where we write the pixels to the ImageData object in the context of the canvas!

This code goes in mandellib.js.

Ready Bake Code Continued...

setUpGraphics sets up the global variables used by all the graphics drawing code as well as the Mandelbrot computation.

```
function setupGraphics() {

    canvas = document.getElementById("fractal");
    ctx = canvas.getContext("2d");

    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    var width = ((i_max - i_min) * canvas.width / canvas.height);
    var r_mid = (r_max + r_min) / 2;
    r_min = r_mid - width/2;
    r_max = r_mid + width/2;

    rowData = ctx.createImageData(canvas.width, 1);

    makePalette();
}
```

Here's where we grab the canvas and the context and set the initial width and height of the canvas.

These are variables used to compute the Mandelbrot Set.

Here, we're initializing the rowData variable (used to write the pixels to the canvas).

And here we're initializing the palette of colors we're using to draw the the set as a fractal image.

This code goes in mandellib.js.

Ready Bake Code Continued...

This **Ready Bake Code** is what the worker will use to do its mathematical computation of the Mandelbrot Set. This is really where the magic of the computation happens (and if you explore the Mandelbrot Set more deeply, this is where you'll want to focus). Type this code into "workerlib.js":

computeRow computes one row of data of the Mandelbrot Set. It's given an object with all the packaged up values it needs to compute that row.

```
function computeRow(task) {
    var iter = 0;
    var c_i = task.i;
    var max_iter = task.max_iter;
    var escape = task.escape * task.escape;
    task.values = [];
    for (var i = 0; i < task.width; i++) {
        var c_r = task.r_min + (task.r_max - task.r_min) * i / task.width;
        var z_r = 0, z_i = 0;

        for (iter = 0; z_r*z_r + z_i*z_i < escape && iter < max_iter; iter++) {
            // z -> z^2 + c
            var tmp = z_r*z_r - z_i*z_i + c_r;
            z_i = 2 * z_r * z_i + c_i;
            z_r = tmp;
        }
        if (iter == max_iter) {
            iter = -1;
        }
        task.values.push(iter);
    }
    return task;
}
```

Notice that for each row of the display, we're doing two loops, one for each pixel in the row...

That's a lot of computation. Good!

... and another loop to find the right value for that pixel. This inner loop is where the computational complexity is, and this is why the code runs so much faster when you have multiple cores on your computer!

The end result of all that computation is a value that gets added to an array of named values, which is put back into the task object so the worker can send the result back to the main code.

We'll take a closer look at this part in a bit.

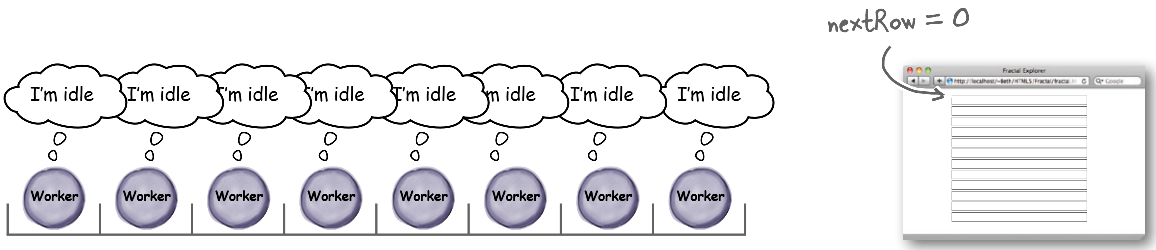
This code goes in workerlib.js.

- [x] Create HTML
- [x] Ready Bake Code
- [ ] Create workers
- [ ] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

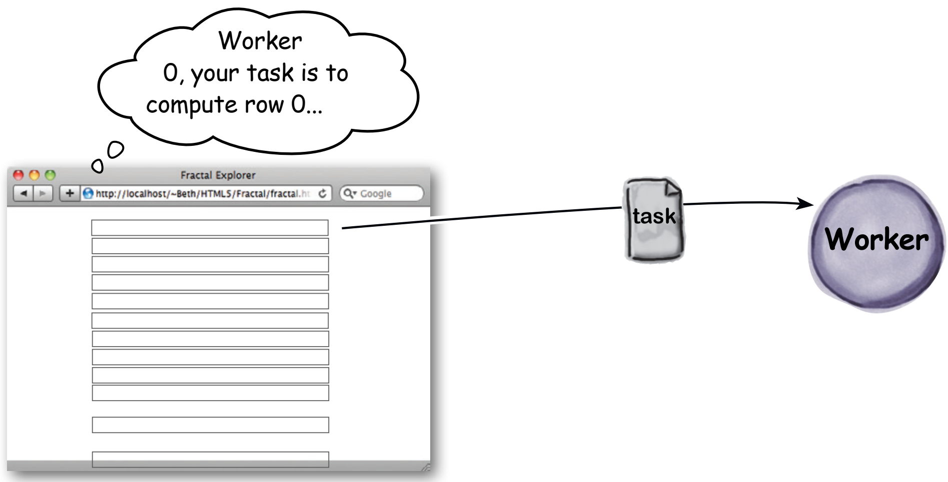
Creating workers, and giving them tasks...

With the **Ready Bake Code** out of the way, let's now turn our attention to writing the code that is going to create and hand tasks to the workers. Here's how it's going to work:

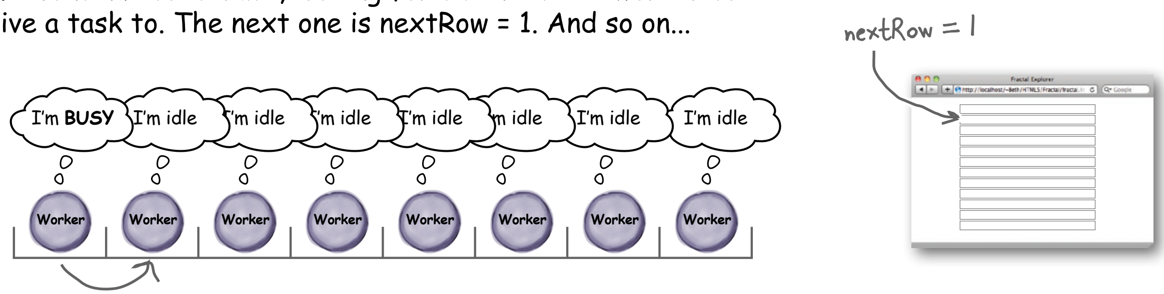
1. We create an array of workers, initially all idle. And an image with nothing computed (nextRow = 0).



2. We iterate through the array, and create a task for each idle worker:



3. We continue to iterate, looking for the next idle worker to give a task to. The next one is nextRow = 1. And so on...



- [x] Create HTML
- [x] Ready Bake Code
- [ ] Create workers
- [ ] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

Writing the code

Now that we know how we're going to create and manage our workers, let's write the code. We really need an initial function for this, so let's create a function in `mandel.js` named `init`—we'll place a few other things in it as well, to get the app up and running (like making sure we have the graphics intialization out of the way):

First let's define a variable that holds the number of workers we want. We're choosing 8, feel free to play with this when you've got the app working.

Why 8? Well, we happen to have a computer with 8 cores so it matches well with our compute power. But even if you don't have 8 cores, 8's a good number to try first.

```
var numberOfWorkers = 8;
var workers = [];

window.onload = init;
```

And here's an empty array to hold our workers.

Let's set up an onload handler that calls init when the page is fully loaded.

```
function init() {
    setupGraphics();

    for (var i = 0; i < numberOfWorkers; i++) {
        var worker = new Worker("worker.js");

        worker.onmessage = function(event) {
            processWork(event.target, event.data);
        }

        worker.idle = true;

        workers.push(worker);
    }

   startWorkers();

}
```

This function is defined in the Ready Bake Code and handles getting the canvas context, resizing the canvas to your browser's size, and a few other graphic details.

Now, iterate over the number of workers...

...and create a new worker from "worker.js", which we haven't written yet.

We then set each worker's message handler to a function that calls the processWork function, and we'll pass it the event.target (the worker that just finished), and the event.data (the results from the worker).

One more thing... remember we are going to want to know which workers are working and which are idle. To do that we'll add an "idle" property to the worker. This is our own property, not part of the Web Worker API. Right now we're setting it to true since we haven't given the workers anything to do.

And we add the worker we just created to the array of workers.

And finally, at some point we need to start these workers working. We'll put that code in a function named startWorkers, which we need to write.

This code goes in mandel.js.

- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [ ] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

Getting the workers started

Okay we've got a few things to knock out: we need to start the workers, we need to write the function that can process the work that comes back from the workers and, well, we need to write the code for the worker too. Let's start by writing the code to start the workers:

We're adding two more global variables to mandel.js.

```
var nextRow = 0;
var generation = 0;

function startWorkers() {
    generation++;
    nextRow = 0;

    for (var i = 0; i < workers.length; i++) {
        var worker = workers[i];

        if (worker.idle) {

            var task = createTask(nextRow);

            worker.idle = false;
            worker.postMessage(task);


            nextRow++;
        }
    }
}
```

The first is nextRow, which keeps track of which row we're on as we work our way through the image.

Every time the user zooms into the Mandelbrot image we start a new image computation. The generation variable keeps track of how many times we've done this. More on this later.

The startWorkers function is going to start the workers, and also restart them if the user zooms into the image. So, each tme we start the workers we reset nextRow to zero and increment generation.

How both of these are used will become clearer in a bit...

Now, we loop over all the workers in the workers array...

... and check to see if the worker is idle.

If it is, we make a task for the worker to do. This task is to compute a row of the Mandelbrot Set. createTask is defined in mandellib.js, and it returns a task object with all the data the worker needs to compute that row.

Now, we're about to give the worker something to do, so we set the idle property to false (meaning, it's busy).

And here's where we tell the worker to start work, by posting a message containing the task. The worker is listening for a message, so when it gets this message, it will start working on the task.

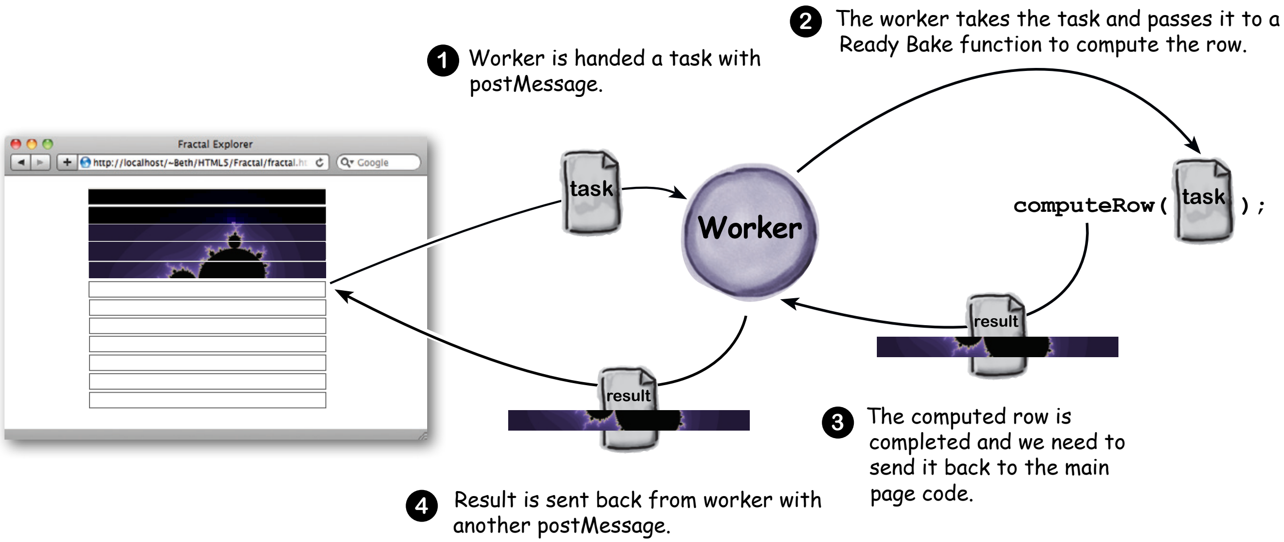
And finally, we increment the row, so the next worker gets the next row.

This code goes in mandel.js.

- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [x] Start the workers
- [ ] Implement the workers
- [ ] Process the results
- [ ] User interaction code

Implementing the worker

Now that we've got the code to get our workers started by passing each a task, let's write the worker code. Then all we need to do is come back and process the results from the worker, once the worker has computed its part of the fractal image. Before we write the code for the worker though, let's quickly review how it should work:



So let's implement this: go head and type the following code into your `worker.js` file.

We're using importScripts to import the workerlib.js Ready Bake Code so the worker can call the computeRow function defined in that library file.

All the worker does is set up the onmessage handler. It doesn't need to do anything else, because all it does is wait for messages from mandel.js to start working!

```
importScripts("workerlib.js");

onmessage = function (task) {

    var workerResult = computeRow(task.data);

    postMessage(workerResult);

}
```

It gets the data from the task, and passes that to the computeRow function, which does the hard work of the Mandelbrot computation.

The result of the computation, saved in the workerResult variable, is posted back to the main JavaScript using postMessage.

This code goes in worker.js

A little pit stop...

That was a lot of code in just a few pages. Let's take a quick pit stop, and refuel our tanks and stomachs.

We also thought you might want to get a quick peek behind the scenes and see what the worker tasks and results look like (they look remarkably similar as we'll see). So, grab a bottle of sarsaparilla and let's take a look while you're resting...

Close up on Tasks

So you've looked at the call to createTask and postMessage, which uses the task:

```
var task = createTask(nextRow);
worker.postMessage(task);
```

And you might be wondering what that task looks like. Well, it's an object made up of propeties and values, let's take a look:

The task contains all the values the worker needs to do its computation.

```
task = {
    row: 1,
    width: 1024,
    generation: 1,
    r_min: 2.074,
    r_max: -3.074,
    i: -0.252336,
    max_iter: 1024,
    escape: 1025
};
```

Identifies the row we're creating the pixels value for.

Identifies width of the row.

Identifies how many times we've zoomed in. We'll see how this is used in a bit...

These define the area of the Mandelbrot we're computing.

And these control the precision of what we're computing.

Close up on Results

And what about the results we get from computing the row in the worker?

```
var workerResult = computeRow(task.data);
postMessage(workerResult);
```

What does this look like? Remarkably similar to the task:

```
workerResult = {
    row: 1,
    width: 1024,
    generation: 1,
    r_min: 2.074,
    r_max: -3.074,
    i: -0.252336,
    max_iter: 1024,
    escape: 1025,
    values: [3, 9, 56, ... -1, 22]
};
```

This is all the same as the task. That's great because when we get it back from the worker we know everything about the task.

The worker takes the task passed to it and then adds a values property to it that contains the data needed to draw the row on the canvas.

Ah, but this is new. These are the values of each pixel, which still need to be mapped to colors (which happens in drawRow).

Time to get back on the road...

Thanks for taking some time with us to check out the tasks and results. You better take a last swig of that sarsaparilla—we're hitting the road again!

- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [x] Start the workers
- [x] Implement the workers
- [ ] Process the results
- [ ] User interaction code

Back to the code: how to process the worker's results

Now that you've seen how the worker's results work, let's see what happens when we get them back from the worker. Recall that when we created our workers, we assigned a message handler named `processWork`:

```
var worker = new Worker("worker.js");

worker.onmessage = function(event) {
   processWork(event.target, event.data);
}
```

Our message handler calls processWork, passing it the data from the worker, and also the target, which is just a reference to the worker that sent the data.

When a worker posts a message back to us with its results, it's the `processWork` function that's going to handle it. As you can see, it is passed two things: the target of the message, which is just a reference to the worker that sent it, and the data of the message (that's the task object with the values for a row of the image). So our job now is to write `processWork` (enter this code in `mandel.js`):

```
function processWork(worker, workerResults) {
     drawRow(workerResults);
     reassignWorker(worker);
}
```

We hand the results to drawRow to draw the pixels to the canvas.

And our worker is all free, so we can reassign it to another task. To do that let's write a function reassignWorker.

We're almost there, so let's just knock out `reassignWorker` while we're at it. Here's how it works: we check the row we're computing by using our `nextRow` global variable, and as long as there's more to compute (which we can determine by looking at how many rows are in our canvas), we give the worker a new assignment. Otherwise, if there's no more work to do, then we just set the worker's idle property to true. Go ahead and enter this code in `mandel.js` too:

```
function reassignWorker(worker) {
     var row = nextRow++;

     if (row >= canvas.height) {
          worker.idle = true;
     } else {
          var task = createTask(row);
          worker.idle = false;
          worker.postMessage(task);
     }
}
```

We're going to give this worker the next row that needs computing, so we get the row number from nextRow, and increment nextRow (so the next worker gets the next one).

If the row is greater than or equal to the height of the canvas, we're done! We've filled the entire canvas with results from the Mandelbrot Set workers.

Canvas is a global variable that was assigned when we called setupGraphics in our init function.

But if we've still got rows to do, we create a new task for the next row to do, make sure our worker's idle property is false, and post a message with the new task to the worker.

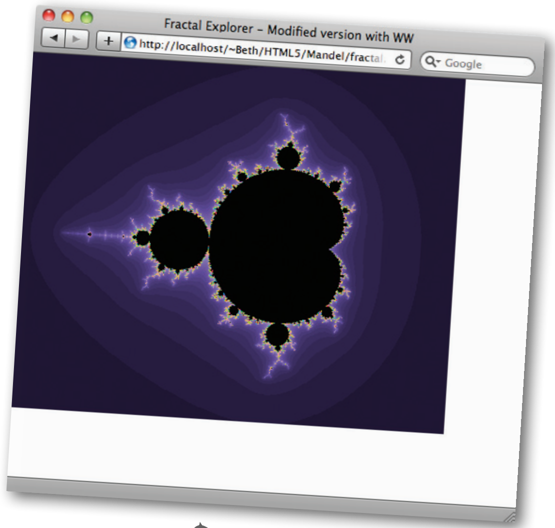
This code goes in mandel.js.

Psychedelic test drive

Enough code already! Let's road test this thing. Load the `fractal.html` file into your browser and see your workers going to work. Depending on your machine, your Fractal Explorer should run a little faster than before.

We haven't written any code to handle resizing your browser window, or clicking to zoom into the fractal for that matter. So, for right now, all you'll be able to see is the image on the right.

But, hey, so far so good, huh?



There it is! Too bad we can't zoom, and too bad it doesn't fill the whole window yet, but we'll get to that...

Handling a click event

We've got our workers busy working to compute the Mandelbrot Set and returning results to us so we can draw them on the canvas, but what happens if you click to zoom in? Fortunately, because we're using workers to do the intense computation in the background, the UI should be snappy in dealing with your click. That said, we need to write a little code to actually handle the click. Here's how we do that:

- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [x] Start the workers
- [x] Implement the workers
- [x] Process the results
- [ ] User interaction code

1. The first thing we need to do is add a handler to take care of mouse clicks, and remember, the clicks are happening on our canvas element. To do that we just add a handler for the canvas's onclick property, like this:

```
canvas.onclick = function(event) {
    handleClick(event.clientX, event.clientY);
};
```

If the canvas is clicked on, we call the function handleClick with the x and y position of the click.

Add this code below the call to setUpGraphics in the init function of "mandel.js".

2. Now we just need to write the handleClick function. Before we do let's think about this a second: when a user clicks on the canvas it means they want to zoom into the area they're clicking on (you can go back to the single-threaded version at `http://wickedlysmart.com/hfhtml5/chapter10/singlethread/fractal.html` to see this behavior). So when the user clicks, we need to get the coordinates of where they want to zoom, and then get all the workers working on creating a new image. Remember too, we've already got a function to assign new work to any idle workers: startWorkers. Let's give it a try...

handleClick is called when the user clicks on the canvas to zoom into the fractal.

We pass in the x, y position of the click so we know where they clicked on the screen.

```
function handleClick(x, y) {
    var width = r_max - r_min;
    var height = i_min - i_max;
    var click_r = r_min + width * x / canvas.width;
    var click_i = i_max + height * y / canvas.height;

    var zoom = 8;

    r_min = click_r - width/zoom;
    r_max = click_r + width/zoom;
    i_max = click_i - height/zoom;
    i_min = click_i + height/zoom;

    startWorkers();
}
```

This code resizes the area of the fractal we are computing, with the x, y position at the center of the new area. It also makes sure the new area has the same aspect ratio of the existing one.

This is where we set the global variables that are used to create tasks for workers: the zoom level determines how far zoomed in we are into the fractal, which determines which values of the Mandelbrot Set are being computed.

Now, we're ready to restart the workers.

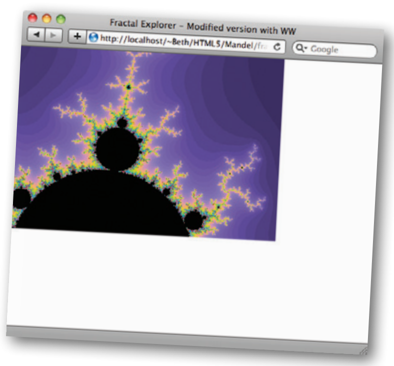
This code goes in mandel.js.

Another test drive

Let's give those code changes a try. Reload `fractal.html` in your browser and this time click somewhere in the canvas. When you do you'll see the workers start working on the zoomed-in view.

Hey, you should be able to start exploring now! After you've played around a bit, let's make a few final changes to get this implementation all the way there.

Nice! We can zoom, but we still need to resize the canvas to fit our window fully.



- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [x] Start the workers
- [x] Process the results
- [x] Process the results
- [ ] User interaction code

Fitting the canvas to the browser window

We said we wanted the fractal image to fill the browser window, which means we need to resize the canvas if the window size changes. Not only that, but if we change the canvas size we should also fire off a new set of tasks to the workers so they can redraw the fractal to fill up the new canvas size. Let's write the code to resize the canvas to the size of the browser window, and we'll also restart the workers while we're at it.

```
function resizeToWindow() {
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;
    var width = ((i_max - i_min) * canvas.width / canvas.height);
    var r_mid = (r_max + r_min) / 2;
    r_min = r_mid - width/2;
    r_max = r_mid + width/2;
    rowData = ctx.createImageData(canvas.width, 1);

    startWorkers();
}
```

resizeToWindow makes sure the canvas width and height are set to match the new width and height of the window.

It also updates the values that the worker will use to do its computation based on the new width and height (we make sure the fractal will always fit the canvas and maintain the aspect ratio of the window).

There's one administrative detail that uses a global variable we haven't told you about: rowData. rowData is the ImageData object that we're using to draw pixels into a row of the canvas. So, when we resize the canvas, we need to recreate the rowData object so that it is the same width as the new width of the canvas. Check the function drawRow in mandellib.js to see how we use rowData to draw pixels into the canvas.

And once again, we restart the workers.

Now we need to do one more thing: install `resizeToWindow` as a handler for the browser window's resize event. Here's how we do that:

```
window.onresize = function() {
   resizeToWindow();
};
```

You'll want to place this code in the init function of `mandel.js`, just below the call to `setUpGraphics`.

This code goes in mandel.js.

The anal-retentive ~~chef~~ coder

There's just one more thing, and we could let this one go, but the code just doesn't seem correct without it. Think through this with us: you've got a bunch of workers happily working on their rows and all of the sudden the user has to go and click on the screen to zoom. Well isn't that great, because the workers have been working hard on their rows, and now the user wants to go and change the entire image, making all that work useless. Even worse, the workers have no knowledge that the user has clicked, and they're going to send back their results anyway. And far worse, the code in the main page is gladly going to receive and display that row! And not to get all doomsday and everything, but we've got exactly the same problem if the user resizes the window.

Note to Editor: Apologies for the little rant here, but, hey, after this many pages, well, it can get to you...

Now, you'd probably never notice any of this because there aren't that many workers, and the workers very quickly compute the same rows for the new image, overwriting the previous, incorrect rows. But hey, it just feels wrong. Not only that, it's so easy to fix we just have to.

Of course we have a little confession to make: we knew this was coming, and you might remember a little variable we stuck in named `generation`. Remember, every time we restart our workers we increase the value of `generation`. Also remember the results object that comes back from the worker: every result has its "generation" as a property. So we can use generation to know if we've got a result from the current or the previous visualization.

Let's look at the code fix, and then we can talk about how it works; edit your `processWork` function in `mandel.js` and add these two lines:

```
function processWork(worker, workerResults) {
    if (workerResults.generation == generation) {
        drawRow(workerResults);
    }
    reassignWorker(worker);
}
```

We're checking the worker's result to see if its generation matches the current one.

If it does match we draw the row, otherwise it must be old and we ignore it.

In either case we get the worker reassigned to new work!

So all we're doing here is checking to make sure the current generation we're working on matches the generation of the result that comes back from the worker. If it does, great, then we need to draw the row. If it doesn't, well that means it must be old, and so we just ignore it—it's too bad our worker wasted its time on it, but we don't want to draw an old row from the previous image on the screen.

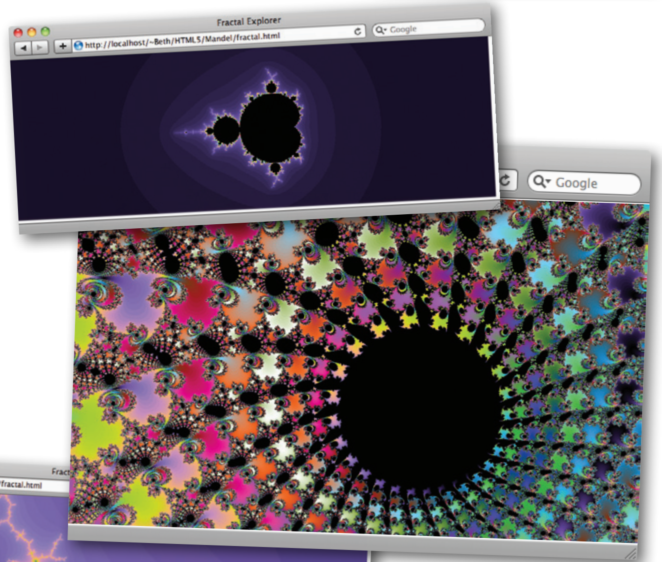
So, really, that's it, we promise, it's time to make sure you have the changes above typed in, and get ready for...

Time for the final test drive! 

- [x] Create HTML
- [x] Ready Bake Code
- [x] Create workers
- [x] Start the workers
- [x] Implement the workers
- [x] Process the results
- [x] User interaction code

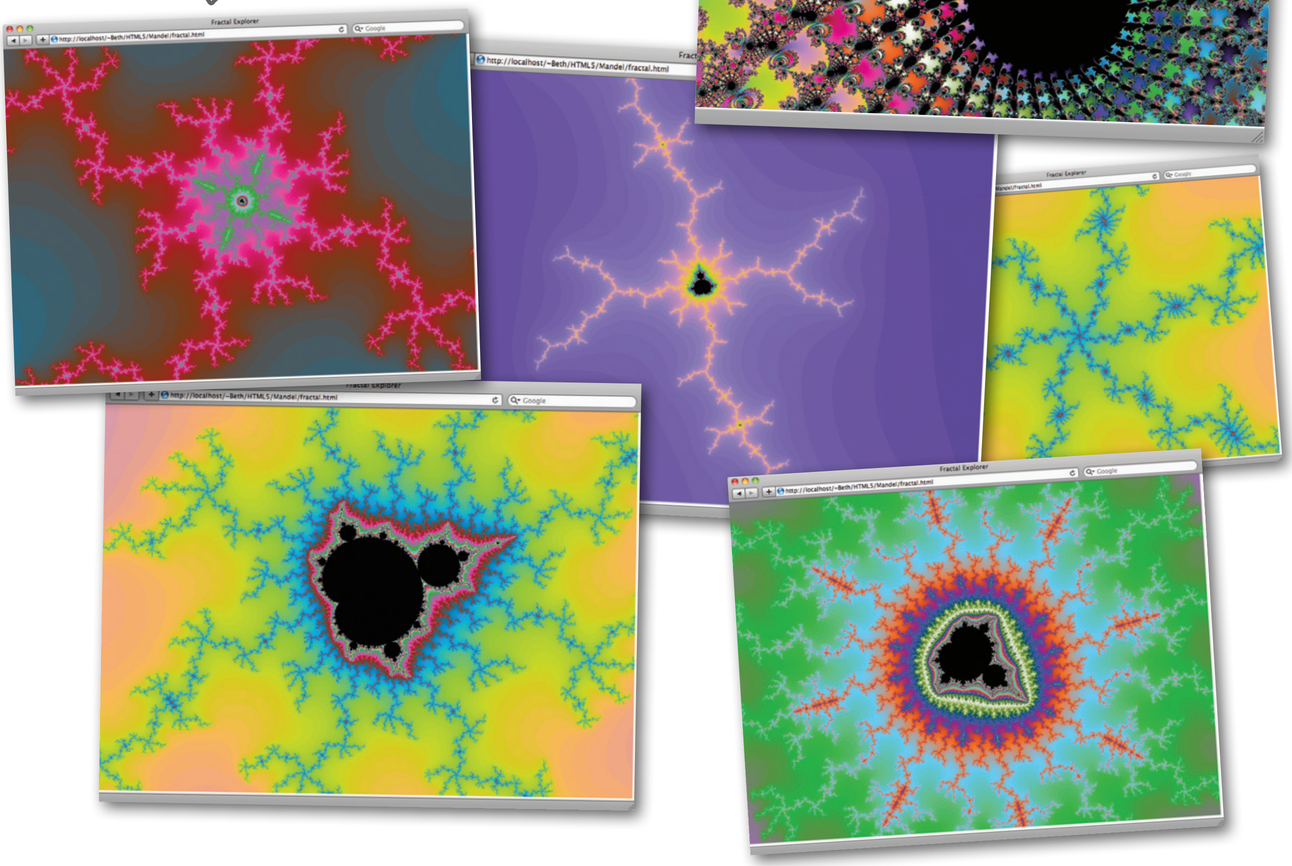
That's it! You should be ready to go with all your code. Load the `fractal.html` file into your browser and see your workers going to work. This version should be faster and more responsive than the original, single-threaded version; if you've got more than one core on your computer, then it will be *a lot* faster.

Have fun... zoom in... explore. Let us know if you find any undiscovered "country" in the Mandelbrot Set (tweet your screenshots to **#hfhtml5** if you want!).



Resize your screen to any shape or size now!

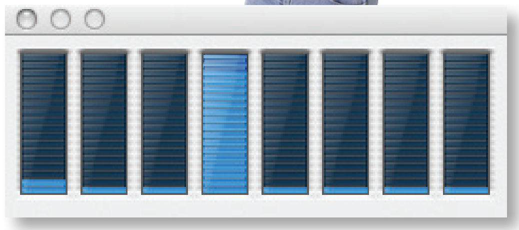
Click, zoom, explore!



IN THE LABORATORY

If you're writing high performance code you'll want to check out how the number of workers can impact your app's runtime.

To do that, you can use the task monitor on either OS X or Windows. If we go back to our original version (the single-threaded one at `http://wickedlysmart.com/hfhtml5/chapter10/singlethread/fractal.html`) our performance looks like the graph on the right.



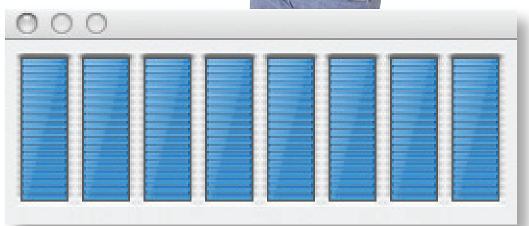
Our machine with eight cores. One core is maxed out and can't compute any harder. The other seven are doing nothing to help.

We have eight cores in our machine, and so in the Fractal Explorer with Web Workers, we're setting the number of workers to match that, with `numberOfWorkers = 8`. And you can see in our activity monitor, all 8 cores are being used to the max.

What do you think will happen if we set the number of workers to 2, or 4, or 16, or 32? Or something in between?

Give it a try on your machine and see what values work best for you.

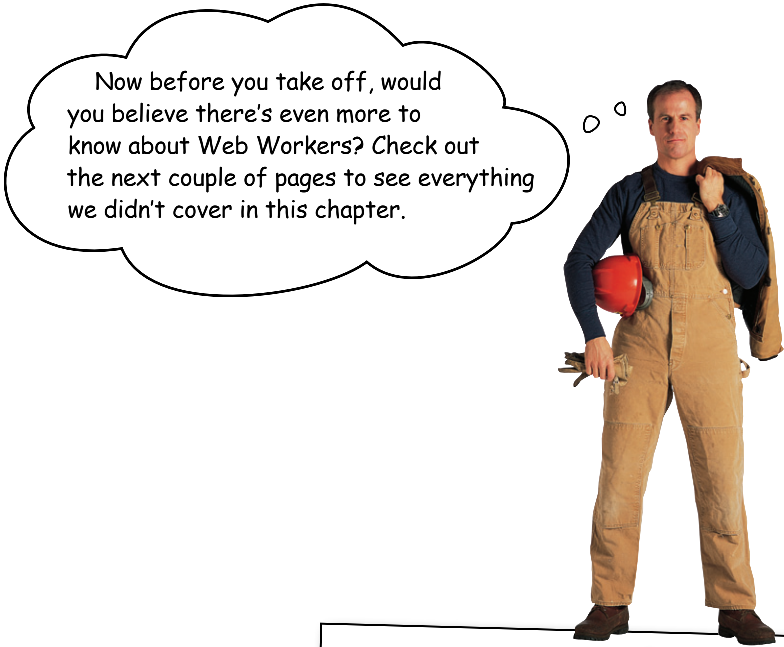
Now our eight cores are really working hard, and our fractal computation is WAY faster.



STAKE YOUR CLAIM!

You've done it! You've got a fully functional Fractal Explorer that's all ready for exploring the Mandelbrot territory. So what are you waiting for—dig in and find your little slice of the virtual universe. Once you've found it, print it, paste it in here, and give your new little homestead a name.

Name your new territory: ______________________



Terminate a worker

You've created workers to do a task, the task is done, and you want to get rid of all the workers (they do take up valuable memory in the browser). You can terminate a worker from the code in your main page like this:

```
worker.terminate();
```

If the worker happens to still be running, the worker script will abort, so use with caution. And once you've terminated a worker you can't reuse it; you'll have to create a new one.

You can also have a worker stop itself by calling **close();** (from inside the worker).

Handle errors in workers

What happens if something goes terribly wrong in a worker? How can you debug it? Use the onerror handler to catch any errors and also get debugging information, like this:

```
worker.onerror = function(error) {
    document.getElementById("output").innerHTML =
        "There was an error in " + error.filename +
        " at line number " + error.lineno +
        ": " + error.message;
}
```

Use importScripts to make a JSONP request

You can't insert new `<script>` elements to make JSONP requests from workers, but you **can** use importScripts to make JSONP requests, like this:

```
function makeServerRequest() {
        importScripts("http://SomeServer.com?callback=handleRequest");
}
function handleRequest(response) {
        postMessage(response);
}
makeServerRequest();
```

Remember your JSONP? Include your callback function in the URL query, and it will be called with the JSON results passed into the response parameter.

Use setInterval in your workers

You might have missed this (it went by fast, we used it in only one example), but you can use setInterval (and setTimeout) in your workers to do the same task repeatedly. For instance you could update the quotes worker (quote.js) to post a random quote every 3 seconds, like this:

```
var quotes = ["I hope life isn't a joke, because I don't get it.",
              "There is a light at the end of every tunnel...just pray it's not a train!",
              "Do you believe in love at first sight or should I walk by again?"];
function postAQuote() {
    var index = Math.floor(Math.random() * quotes.length);
    postMessage(quotes[index]);
}
postAQuote();
setInterval(postAQuote, 3000);
```

Move these two lines into a postAQuote function...

... call postAQuote to send a quote right away, and then set an interval to send more quotes, every 3 seconds.

Subworkers

If your worker needs help with its task, it can create its own workers. Say you're giving your worker regions of an image to work on, the worker could decide that if a region is bigger than some size, it will split it up among its own subworkers.

A worker creates subworkers just like the code in your page creates a worker, with:

```
var worker = new Worker("subworker.js");
```

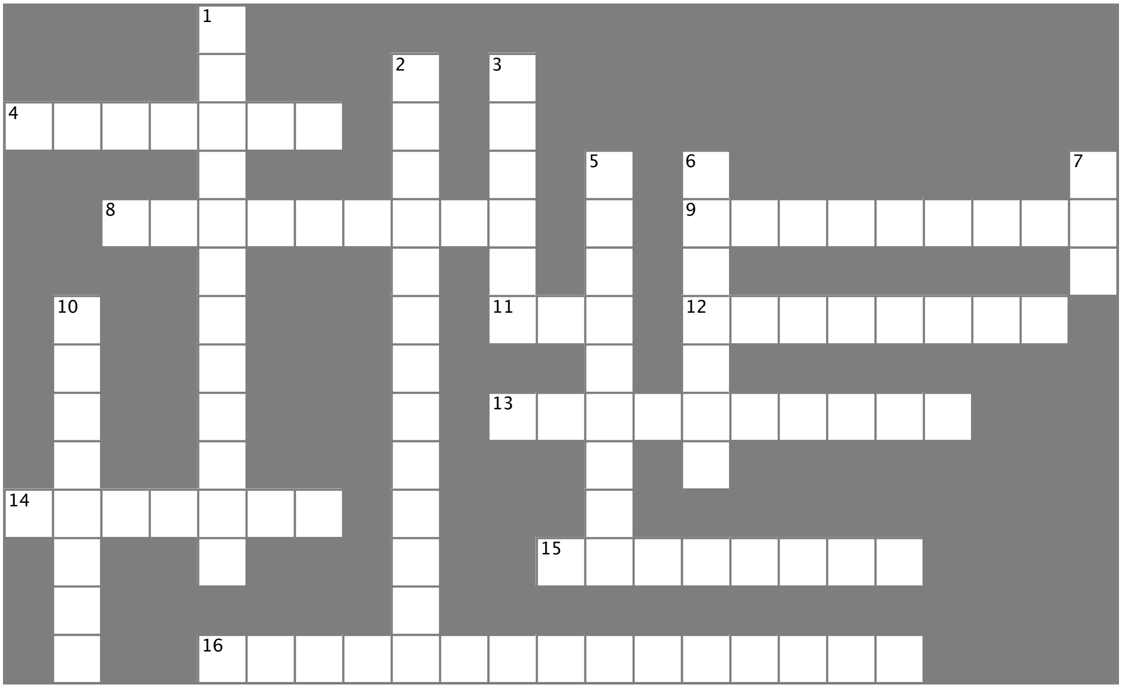
Remember that subworkers, just like workers, are fairly heavy-weight: they take up memory and are run as separate threads. So, be cautious about how many subworkers you create.

BULLET POINTS

- Without Web Workers, JavaScript is single-threaded, meaning it can do only one thing at a time.
- If you give a JavaScript program too much to do, you might get the slow script dialog.
- Web Workers handle tasks on a separate thread so your main JavaScript code can continue to run and your UI remains responsive.
- The code for a Web Worker is in a separate file from your page's code.
- Web Workers don't have access to any of the functions in the code in your page or the DOM.
- The code in your page and the Web Worker communicate via messages.
- To send a message to a worker, use postMessage.
- You can send strings and objects to a worker via postMessage. You can't send functions to a worker.
- Receive messages back from workers by setting the worker's onmessage property to a handler function.
- A worker receives messages from the code in your page by setting its onmessage property to a handler function.
- When a worker is ready to send back a result, it calls postMessage and passes the result as the argument.
- Worker results are encapsulated in an event object and placed in the data property.
- You can find out which worker sent the message using the event.target property.
- Messages are copied, not shared, between your main page code and the worker.
- You can use multiple workers for large computations that can be split into multiple tasks, such as computing a fractal visualization or ray tracing an image.
- Each worker runs in its own thread, so if your computer has a multicore processor, the workers are run in parallel, which increases the speed of the computation.
- You can terminate a worker by calling worker.terminate() from the code in your page. This will abort the worker script. A worker can also stop itself by calling close().
- Workers also have an onerror property. You can set this to an error handling function that will be called if your worker has a script error.
- To include and use JavaScript libraries in your worker file, use importScripts.
- You can also use importScripts with JSONP. Implement the callback you pass in the URL query in the worker file.
- While workers do not have access to the DOM or functions in your main code, they can use XMLHttpRequest and Local Storage.

HTML5cross

Wow, Chapter 10; you've done it. Sit back, relax and make it stick by working the rest of your brain a little. Here's your Chapter 10 crossword puzzle.



Across

4. You can pass _______ to workers using postMessage.
8. Capability of a processor to do more than one thing at a time.
9. The property used to register a handler to receive messages.
11. Workers can't access the _______.
12. Our first example used this game.
13. The most famous fractal.
14. ______/worker.
15. A lovely area of the Mandelbrot countryside is _________ Valley.
16. The guy who wrote the original version of Fractal Viewer.

Down

1. Workers can use XMLHttpRequest and access ___________.
2. How to import additional code into a worker.
3. _____ of execution.
5. How to abort a worker.
6. Mandelbrot uses __________ numbers.
7. How to create a Worker.
10. The manager and workers communicate with these.

Okay we never told you this,
it's James Henstridge.

BE the Browser Solution

It's time to pretend you're the browser evaluating JavaScript.

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
    }
    for (var i = 0; i < 5; i++) {
        worker.postMessage("ping");
    }
}
```

This sends five ping messages to the worker, which responds with five pongs, so we get five "Worker says pong" alerts.

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.ommessage = function(event) {
        alert("Worker says " + event.data);
    }
    for(var i = 5; i > 0; i--) {
        worker.postMessage("pong");
    }
}
```

This sends five pong messages to the worker, which ignores them since they aren't pings. No output.

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
        worker.postMessage("ping");
    }
    worker.postMessage("ping");
}
```

This sends a ping and then each time a pong comes back, sends another, so we get an infinite loop of pong alerts.

```
window.onload = function() {
    var worker = new Worker("worker.js");
    worker.onmessage = function(event) {
        alert("Worker says " + event.data);
    }

    setInterval(pinger, 1000);

    function pinger() {
        worker.postMessage("ping");
    }
}
```

This sends a ping every second, so we get a pong back each time it sends a ping.

Sharpen your pencil Solution

While workers typically get their work orders through a message, they don't have to. Check out this nice, compact way to get work done with workers and HTML. When you know what it does, describe it below:

quote.html

```
<!doctype html>
<html lang="en">
   <head>
      <title>Quote</title>
      <meta charset="utf-8">
   </head>
<body>
   <p id="quote"></p>
   <script>
       var worker = new Worker("quote.js");
       worker.onmessage = function(event) {
           document.getElementById("quote").innerHTML = event.data;
       }
   </script>
</body>
</html>
```

quote.js

```
var quotes = ["I hope life isn't a joke, because I don't get it.",
              "There is a light at the end of every tunnel....just pray it's not a train!",
              "Do you believe in love at first sight or should I walk by again?"];
var index = Math.floor(Math.random() * quotes.length);

postMessage(quotes[index]);
```

Your description here:

In our HTML, we have a script that creates a worker, which executes immediately. The worker chooses a quote randomly from the quotes array, and sends the quote to the main code using postMessage. The main code gets the quote from event.data and adds it to the page in the "quote" <p> element.

HTML5cross Solution

				1 L																		
				O				2 I		3 T												
4 O	B	J	E	C	T	S		M		H												
				A				P		R		5 T		6 C								7 N
		8 M	U	L	T	I	C	O	R	E		E		9 O	N	M	E	S	S	A	G	E
				S				R		A		R		M								W
	10 M			T				T		11 D	O	M		12 P	I	N	G	P	O	N	G	
	E			O				S				I		L								
	S			R				C		13 M	A	N	D	E	L	B	R	O	T			
	S			A				R				A		X								
14 M	A	N	A	G	E	R		I				T										
	G			E				P			15 S	E	A	H	O	R	S	E				
	E							T														
	S			16 J	A	M	E	S	H	E	N	S	T	R	I	D	G	E				

Congratulations!
You made it to the end.

Of course, there's still an appendix.

And the index.

And the colophon.

And then there's the web site...

There's no escape, really.

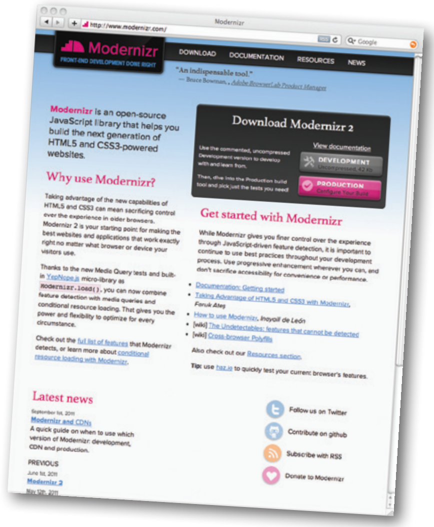
Appendix: leftovers

The Top Ten Topics (we didn't cover)

We've covered a lot of ground, and you're almost finished with this book. We'll miss you, but before we let you go, we wouldn't feel right about sending you out into the world without a little more preparation. We can't possibly fit everything you'll need to know into this relatively small chapter. Actually, we *did* originally include everything you need to know about HTML5 (not already covered by the other chapters), by reducing the type point size to .00004. It all fit, but nobody could read it. So, we threw most of it away, and kept the best bits for this Top Ten appendix.

This really *is* the end of the book. Except for the index, of course (a must-read!).

#1 Modernizr



One thing you've probably noticed in this book, is that when you want to detect browser support for an API, there is no uniform way of doing so; in fact, almost every API is detected in a different way. For geolocation, for instance, we look for the geolocation object as a property of the navigator object, while for web storage we check to see if localStorage is defined in the window object, and for video we check to see if we can create a video element in DOM, and so on. Surely there's a better way?

Modernizr is an open source JavaScript library that provides a uniform interface for detecting browser support. Modernizer takes care of all the details of the different means of detection, even factoring in all the edge cases around older browsers. You'll find the Modernizr home page at `http://www.modernizr.com/`

Modernizr has gained a lot of developer support so you'll see it used widely around the Web. We highly recommend it.

Including Modernizr in your page

To use Modernizr, you need to load the JavaScript library into your page. To do that you first visit the Modernizer site at `http://www.modernizr.com/download/`, which allows you to custom configure a library that contains just the detection code you need (or you can always grab everything while you're there). After you've done that, stash the library in a file of your choice and load it into your page (visit Modernizr's web site for addition tutorials and documentation on best practices for doing this).

How to detect support

Once you've got Modernizr installed, detecting HTML5 elements and JavaScript APIs gets a lot easier and more straightforward:

Here's an example of detecting for geolocation, web storage and video, all in a consistent manner.

Note: Modernizr goes far beyond simple API detection and can also detect support for CSS features, video codecs and many other things. So, check it out!

```
if (Modernizr.geolocation) {
    console.log("You have geo!");
}
if (Modernizr.localstorage) {
    console.log("You have web storage!");
}
if (Modernizr.video) {
    console.log("You have video!");
}
```

#2 Audio

HTML5 gives you a standard way to play audio in your pages, without a plug-in, with the `<audio>` element:

```
<audio src="song.mp3" id="boombox" controls>
  Sorry but audio is not supported in your browser.
</audio>
```

Look familiar? Yes, audio supports similar functionality as video (minus video, obviously).

In addition to the `<audio>` element, there is also a corresponding Audio API that supports the methods you'd expect, like `play`, `pause` and `load`. If this sounds familiar, it should, because the audio API mirrors (where appropriate) the video API. Audio also supports many of the properties you saw in the video API, like `src`, `currentTime` and `volume`. Here's a bit of audio code to get a feel for using the API with an element in the page:

```
var audioElement =
    document.getElementById("boombox");

audioElement.volume = .5;

audioElement.play();
```

Get a reference to the audio element, then lower its volume to 1/2 and start playing.

Also like video, each browser implements its own look and feel for player controls (which typically consist of a progress bar with play, pause and volume controls).

Despite its simple functionality, the audio element and API give you lots of control. Just like we did with video, you can create interesting web experiences by hiding the controls and managing the audio playback in your code. And with HTML5, you can now do this without the overhead of having to use (and learn) a plug-in.

A Standard for Audio Encodings

Sadly, like video, there is no standard encoding for audio. Three formats are popular: mp3, wav and Ogg Vorbis. You'll find that support for these formats varies across the browser landscape with different levels of support for the various formats in each browser (as of this writing, as of this writing, Chrome is the only browser that supports all three formats).

#3 jQuery

jQuery is a JavaScript library that is aimed at reducing and simplifying much of the JavaScript code and syntax that is needed to work with the DOM, use Ajax and add visual effects to your pages. jQuery is an enormously popular library that is widely used and expandable through its plug-in model.

Remember, Ajax is just a name for using XMLHttpRequest, like we did in Chapter 6.

Now, there's nothing you can do in jQuery that you can't do with JavaScript (as we said, jQuery is just a JavaScript library), however it does have the power to reduce the amount of code you need to write.

jQuery's popularity speaks for itself, although it can take some getting used to if you are new to it. Let's check out a few things you can do in jQuery and we encourage you to take a closer look if you think it might be for you.

A working knowledge of jQuery is a good skill these days on the job front and for understanding others' code.

For starters, remember all the window onload functions we wrote in this book? Like:

```
window.onload = function() {
    alert("the page is loaded!");
}
```

Here's the same thing using jQuery:

```
$(document).ready(function() {
    alert("the page is loaded!");
});
```

Just like our version, when the document is ready, invoke my function.

Or you can shorten this even more, to:

```
$(function() {
    alert("the page is loaded!");
});
```

This is cool, but as you can see it takes a little getting used to at first. No worries, it becomes second-nature fast.

So what about getting elements from the DOM? That's where jQuery shines. Let's say you have an anchor in your page with an id of "buynow" and you want to assign a click handler to the click event on that element (like we've done a few times in this book). Here's how you do that:

So what's going on here? First we're setting up a function that is called when the page is loaded.

```
$(function() {
  $("#buynow").click(function() {
      alert("I want to buy now!");
  });
});
```

Next we're grabbing the anchor with a "buynow" id (notice jQuery uses CSS syntax for selecting elements).

And then we're calling a jQuery method, click, on the result to set the onclick handler.

That's really just the beginning; we can just as easily set the click handler on *every anchor* on the page:

```
$(function() {
   $("a").click(function() {
       alert("I want to buy now!");
   });
});
```

To do that, all we need to do is use the tag name.

Compare this to the code you'd write to do this if we were using JavaScript without jQUery.

Or, we can do things that are much more complex:

Like find all the <li> elements that are children of the element with an id of playlist.

```
$(function() {
   $("#playlist > li").addClass("favorite");
});
```

And then add them to the class "favorite".

Actually this is jQuery just getting warmed up; jQuery can do things much much more sophisticated than this.

There's a whole 'nother side of jQuery that allows you to do interesting interface transfomations on your elements, like this:

```
$(function() {
    $("#specialoffer").toggle(function() {
        $(this).animate({ backgroundColor: "yellow" }, 800);
    },function() {
        $(this).animate({ backgroundColor: "white" }, 300);
    });
});
```

This toggles the element with an id of specialoffer between being yellow and 800 pixels wide, and white and 300 pixels wide, and animates the transition between the two states.

As you can see, there's a lot you can do with jQuery, and we haven't even talked about how we can use jQuery to talk to web services, or all the plug-ins that work with jQuery. If you're interested, the best thing you can do is point your browser to `http://jquery.com/` and check out the tutorials and documentation there.

And, check out Head First jQuery too!

#4 XHTML is dead, long live XHTML

We were pretty tough on XHTML in this book, first with the "XHTML is dead" discussion, and then later with "JSON versus XML". The truth is, when it comes to XHTML, it is only XHTML 2 and later that has died, and in fact, you can write your HTML5 using XHTML-style if you want to. Why would you want to? Well, you might need to validate or transform your documents as XML, or you might want to support XML technologies, like SVG (see #5), that work with HTML.

Let's look at a simple XHTML document and then step through the high points (we couldn't possibly cover everything you need to know on this topic, as with all things XML; it gets complicated, fast).

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
   <head>
      <title>You Rock!</title>
      <meta charset="UTF-8" />
   </head>
   <body>
      <p>I'm kinda liking this XHTML!</p>
      <svg xmlns="http://www.w3.org/2000/svg">
         <rect stroke="black" fill="blue" x="45px" y="45px"
               width="200px" height="100px" stroke-width="2" />
      </svg>
   </body>
</html>
```

Same doctype!

This is XML, we need a namespace!

All elements have to be well formed; note the trailing /> here to close this empty element.

We're using SVG to draw a rectangle into our page. Check out #5 (next page) for more on SVG.

We can embed XML right in the the page! Kinda cool.

Now here's a few things you need to consider for your XHTML pages:

Closing all your elements, quotes around attribute values, valid nesting of elements, and all that.

- Your page must be well formed XML.
- Your page should be served with the `application/xhtml+xml` MIME type, for this you'll need to make sure your server is serving this type (either read up on this or contact your server administrator).
- Make sure and include the XHTML namespace in your `<html>` element (which we've done above).

Like we said, with XML there's a lot more to know and lots of things to watch out for. And, as always with XML, may the force be with you...

#5 SVG

Scalable Vector Graphics, or SVG, is another way—aside from canvas—of including graphics natively in your web pages. SVG has been around a while (since 1999 or so) and is now supported in all the current versions of major browsers, including IE9 and later.

Unlike canvas, which, as you know, is an element that allows you to draw pixels into a bitmap drawing surface in your page with JavaScript, SVG graphics are specified with XML. "XML?" you say? Yes, XML! You create elements that represent graphics, and then you can combine those elements together in complex ways to make graphic scenes. Let's take a look at a very simple SVG example:

We're using the XHTML-style HTML5 because we're using SVG, which is XML-based.

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>SVG</title>
  <meta charset="utf-8" />
</head>
<body>
  <div id="svg">
    <svg xmlns="http://www.w3.org/2000/svg">
      <circle id="circle"
              cx="50" cy="50" r="20"
              stroke="#373737" stroke-width="2"
              fill="#7d7d7d" />
    </svg>
  </div>
</body>
</html>
```

We're using an <svg> element right in our HTML!

Our SVG is simple: it contains only a circle that is located at position x=50, y=50 and has a radius of 20...

... a stroke that is 2 pixels wide and colored dark grey...

... and is filled with a medium grey.

You can grab this circle element just like any other element from the DOM and do stuff with it... for instance you could add a click handler and change the circle's fill attribute to "red" when the user clicks on the circle.

SVG defines a variety of basic shapes, like circles, rectangles, polygons, lines, and so on. If you have more complex shapes to draw, you can also specify paths with SVG—of course, at that point things start getting more complex (as you already saw with paths in canvas). However, there are graphical editors that will let you draw a scene and export it as SVG, saving you the headache of figuring out all those paths yourself!

What's so great about SVG? Well, one nice aspect of SVG is that you can scale your graphics as big or small as you want and they don't pixellate, like a jpeg or png image would if you scaled it. That makes them easy to reuse in different situations. And because SVG is specified with text, SVG files can be searched, indexed, scripted and compressed.

We've barely scratched the surface of what you can do with SVG, so explore more if this topic interests you.

#6 Offline web apps

If you've got a smartphone or tablet, you're probably accessing the Web on the go, and with WiFi and cellular networks, you're connected almost all the time. But what about those times when you're not? Wouldn't it be great if you could keep on using those great HTML5 web apps you've been building for yourself?

Well, now you can. Offline web applications are supported by all modern desktop and mobile browsers (with one exception: IE).

So how do you make your web application available offline? You create a *cache manifest* file that contains a list of all the files your app needs to work, and the browser will download all those files, and switch to the local files if and when your device goes offline. To tell your web page that it has a manifest file, you simply add the filename of the cache manifest file to your `<html>` tag, like this:

```
<html manifest="notetoself.manifest">
```

Here's what the `notetoself.manifest` file contains:

```
CACHE MANIFEST
CACHE:
notetoself.html
notetoself.css
notetoself.js
```

Every cache manifest file must start with this.

List all the files you want to cache in the CACHE section: html, css, javascript, images, etc.

With offline web apps, you can use your favorite web apps when you're not connected!

This file says: when you visit the web page that points to this file, download all the files listed in the `CACHE` section of the file. You can also add two other sections to the file, `FALLBACK` and `NETWORK`. `FALLBACK` specifies what file to use if you try to access a file that isn't cached, and `NETWORK` specifies files that should never be cached (for example, visit tracking resources).

Now, before you run off to go play with this, you need to know two things: first, you need to make sure your web server is set up to serve the mime type for cache manifest files correctly (just like we had to do for video files in Chapter 8). For example, on an Apache server, add this line to your `.htaccess` file at the top level of your web directory:

```
AddType text/cache-manifest .manifest
```

The other thing you need to know is that testing offline web applications is tricky! We recommend checking out a good reference on the topic and reading the HTML5 offline web applications specification.

Once you've got basic caching working, you can use JavaScript to be notified of cache events, such as when a cache manifest file is updated and the status of the cache. To be notified of events, you add event handlers to the `window.applicationCache` object, like this:

```
window.applicationCache.addEventListener("error", errorHandler, false);
```

Implement the errorHandler to be notified if there's an error with the cache.

#7 Web Sockets

We looked at two ways of communicating in this book: XMLHttpRequest and JSONP. In both cases we used a request/response model based on HTTP. That is, we used the browser to make a request for the initial web page, CSS and JavaScript, and each time we needed something else, we made another request using XMLHttpRequest or JSONP. We even made requests when there was no new data for us, which happened sometimes in the Mighty Gumball example.

Web Sockets is a new API that allows you to keep an open connection with a web service so that any time new data is available the service can just send it to you (and your code can be notified). Think of it like an open phone-line between you and the service.

Here's a high-level overview of how you use it: first, to create a web socket we use the web socket constructor:

Notice this URL uses the ws protocol, not the http protocol.

```
var socket = new WebSocket("ws://yourdomain/yourservice");
```

And remember you or someone else is going to have to write the server code so you have something to talk to!

You can be notified as soon as the socket is open with the `open` event, which you can assign a handler for:

```
socket.onopen = function(){
    alert("Your socket is now open with the web service");
}
```

Here we supply a handler that is called when the socket is fully opened and ready for communcation.

You can send a message to the web service with the `postMessage` method:

```
socket.postMessage("player moved right");
```

Here's we're sending the server a string; binary is coming but not widely supported yet.

And to receive messages you register another handler, like this:

```
socket.onmessage = function(event) {
    alert("From socket: " + event.data);
};
```

By registering a handler, we receive all messages, which are contained in the event's data property.

There's a little more to it than this, of course, and you'll want to check out some tutorials online, but there's not much more to the API. This API has been lagging behind some of the other HTML5 API development, so check out the latest browser compatibility guides before you undertake a major project.

#8 More canvas API

We had fun with the canvas in Chapter 7, building our TweetShirt startup. But there are lots of other fun canvas-related things you can do and we wanted to touch on a few more of them here.

We mentioned very briefly that you can `save` and `restore` the canvas context. Why would you want to do that? Let's say you've set some properties of the context, like the `fillStyle`, `strokeStyle`, `lineWidth` and so on. And you want to then temporarily change those values to do one thing, like draw a shape, but not have to reset them all to get back to the property values you had previously. You can use the `save` and `restore` methods to do that:

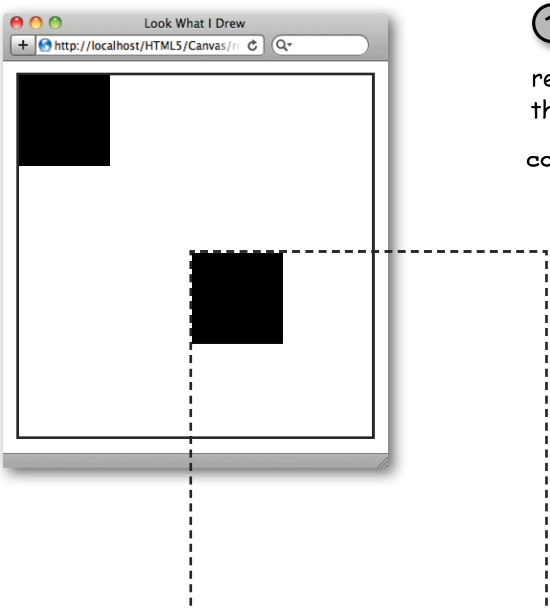
```
context.fillStyle = "lightblue";
 ...
context.save();
context.fillStyle = "rgba(50, 50, 50, .5)";
context.fillRect(0, 0, 100, 100);
context.restore();
 ...
```

We set up a bunch of properties in the context and do some drawing.

Now, we save the context. All those properties are saved safely. We can change them...

... and then get them all back to where they were when we saved them simply by calling the restore method! At this point, all our properties are what they were before we saved.

These methods come in particularly handy when you want to *translate* or *rotate* the canvas to draw something and then put it back to its default position. What do the `translate` and `rotate` methods do? Let's take a look...

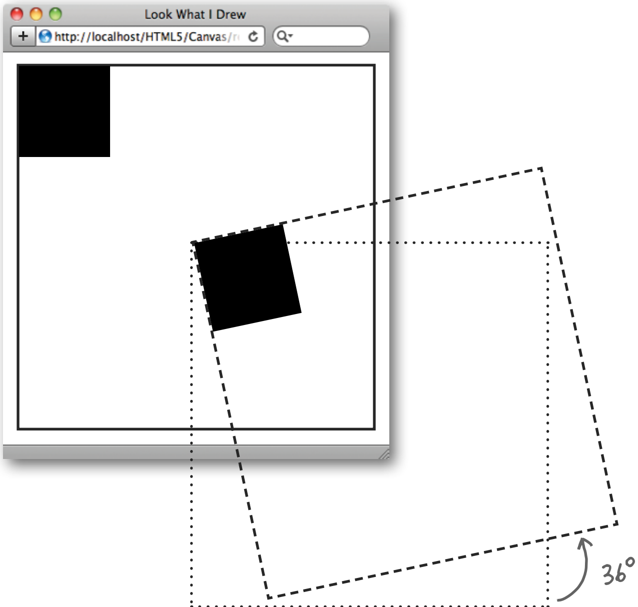


① We've got a 400x400 canvas in the page. If we draw a black rectangle at x=0, y=0, it is drawn in the top left corner, like you'd expect.

```
context.fillRect(0, 0, 100, 100);
```

② Now, we'll pick up the canvas and move it 200 pixels to the right, and 200 pixels down. If we draw another rectangle at x=0, y=0, the rectangle is drawn 200 pixels right and down from the other rectangle. We've just translated the canvas.

```
context.translate(200, 200);
context.fillRect(0, 0, 100, 100);
```



3. What if we rotate the canvas before we draw the rectangle? The canvas rotates around its top left corner (by default), and since we just moved the top left corner to 200, 200, that's the position where the canvas rotates.

```
context.translate(200, 200);
context.rotate(degreesToRadians(36));
context.fillRect(0, 0, 100, 100);
```

When you translate or rotate the canvas, it's moved on a grid that's positioned with respect to the top left corner of the browser window. If you've positioned your canvas using CSS, those values are taken into account. Try it!

Now let's put all that together! You can use the `translate` and `rotate` methods together to create some interesting effects.

```
var canvas = document.getElementById("canvas");
var context = canvas.getContext("2d");
var degrees = 36;
```

We're saving the context here so we can easily restore it to its normal grid position after we're done.

```
context.save();
context.translate(200, 200);
```

We translate our canvas by 200, 200.

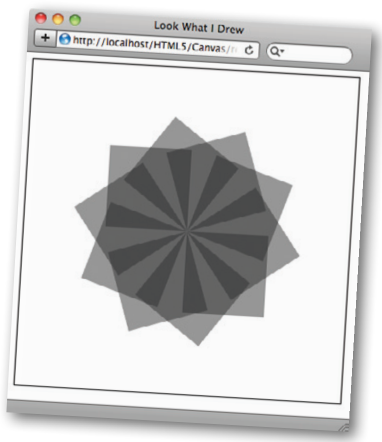
```
context.fillStyle = "rgba(50, 50, 50, .5)";

for (var i = 0; i < 360/degrees; i++) {
  context.fillRect(0, 0, 100, 100);
  context.rotate(degreesToRadians(degrees));
}
```

We're drawing 10 rectangles by rotating the canvas 36 degrees before drawing a rectangle at 0, 0 each time through the loop.

```
context.restore();
```

Now our canvas is back at its original position!

And here's the result. Fun!



Combine these simple transformations with other, even more powerful (and complex!) methods like compositing and transforms, and the possibilities for creating graphic art with canvas are endless.

#9 Selectors API

You already know how to select elements from the DOM using `document.getElementById`; we've used it throughout this book as a way to get HTML and JavaScript working together. You've also seen how to use `document.getElementsByTagName` (this method returns an array of all elements that match a tag), and there's even a `getElementsByClassName` method (returning, you guessed it, all elements that are in a given class).

With HTML5, we now have a new way of selecting elements from the DOM, inspired by jQuery. You can now use the same selectors you use in CSS to select elements for styling in your JavaScript to select elements from DOM with the `document.querySelector` method.

Let's say we have this simple HTML:

```
<!doctype html>
<html lang="en">
<head>
  <title>Query selectors</title>
  <meta charset="utf-8">
</head>
<body>
  <div class="content">
    <p id="avatar" class="level5">Gorilla</p>
    <p id="color">Purple</p>
  </div>
</body>
</html>
```

Take a close look at the structure of this HTML. We're going to use the selectors API to select elements from the page.

We've got a <div> element with the class "content", and two <p> elements, each with their own ids, and one with the class "level5".

Now, let's use the selectors API to ask for the "avatar" <p> element:

```
document.querySelector("#avatar");
```

That's essentially the same thing as `document.getElementId("avatar")`. Now let's use the element's class to select it:

```
document.querySelector("p.level5");
```

Now we're using the tag name and the class to select it.

We can also select a <p> element that is a child of the `<div>` element, like this:

```
document.querySelector("div>p");
```

or even like this:

```
document.querySelector(".content>p");
```

Here, we're using a child selector to select a <p> element that is a child of the <div>. It selects the first one, by default.

And, if what we really want is *all* the <p> elements in the `<div>`, we can use the other method in the selectors API, `querySelectorAll`:

```
document.querySelectorAll("div>p");
```

Now we get all the child <p> elements of the <div>!

`querySelectorAll` returns an array of elements, just like `getElementsByTagName`. And that's it! Those are the only two methods in the API. The selectors API is small, but adds powerful new functionality for selecting elements.

#10 But, there's even more!

Okay, we really wanted to keep it to the ten things we didn't tell you, but it looks like we've got a ways to go, and instead of standing between you and your reading of the index, we're going to give you a bunch more in a single page. Here they are (keep in mind some of these areas are still evolving, but we knew you'd want to know about them for future reference):

Indexed Database API and Web SQL

If you're looking for something more industrial than the Web Storage API to store your data locally, keep an eye on the web database space. Two competing visions are out there right now: Web SQL and IndexedDB. Ironically, Web SQL is the more widely supported of the two, but was recently deprecated by the standards bodies (meaning they don't recommend adopting it as a standard, and you probably shouldn't base your next startup on it). IndexedDB, on the other hand, isn't widely implemented yet, but has support from Google and Firefox. IndexedDB provides fast access to a large collection of indexed data, while Web SQL is a small SQL engine that runs in the browser. Keep an eye out for where these technologies go; they are changing fast!

Drag and Drop

Web developers have been doing drag and drop with jQuery for a while now, and now this functionality is native in HTML5. With the HTML5 Drag and Drop API, you specify something to drag, where you can drop it, and JavaScript handlers to be notified of the various events that occur while dragging and dropping. To make an element draggable, just set the `draggable` attribute to true. Just about any element can be dragged: images, lists, paragraphs, and so on. You can customize the dragging behavior by listening for events like `dragstart` and `dragend` and even change the style of an element to look like you want while it's being dragged. You can send along a little bit of data with your dragged element using the `dataTransfer` property; access this through the `event` object to know if, say, the element is being moved or copied. As you can see, there are lots of great opportunities to build new UI interactions with HTML5 Drag and Drop.

Cross-document Messaging

In chapter 6, we used a communication pattern known as JSONP to get around the cross-domain communication issues with XMLHttpRequest. There's another way you can communicate between documents—even documents in different domains. The Cross-document Messaging API specifies that you can post a message to a document you've loaded using an `iframe` element. This document could even be at a different domain! Now, you wouldn't want to load just *any* document into your `iframe`; you'll want to make sure it's from a domain you trust and set it up to receive your messages. But the upshot is that this is a way to get messages back and forth between two HTML documents.

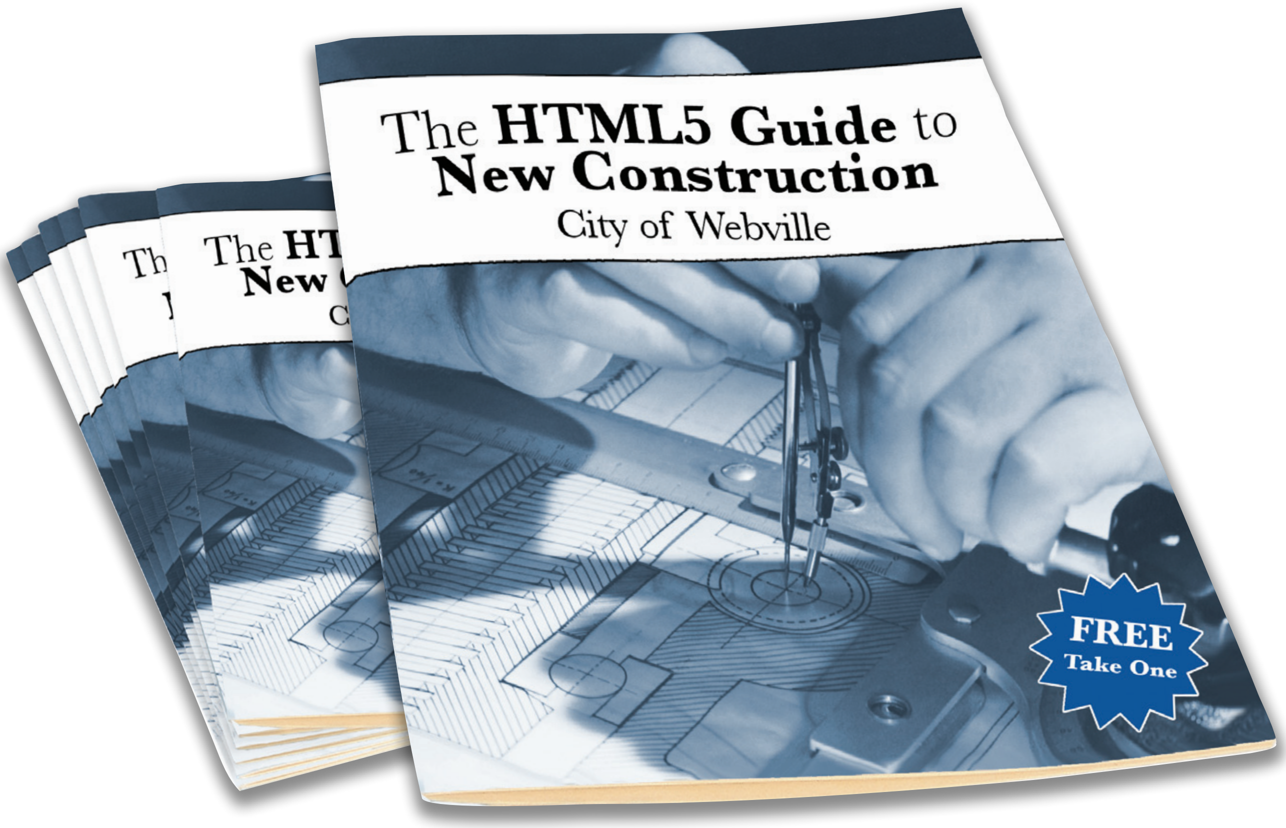
And we could go on...

The exciting thing about HTML5 is that there are so many new capabilities being developed at a fairly fast pace; there's even more we could put on the this page, but we're out of room. So keep up to date with us on the Web at `http://wickedlysmart.com` for all the latest developments in HTML5!

I can't believe the book is almost over. Before you go, we've got a little parting gift for you from the City of Webville; it's the guide to the HTML5 elements (and what's new in CSS3) that we promised you. Isn't Webville great?!

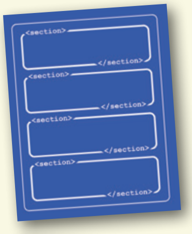
The HTML5 Guide to New Construction

Here in Webville we've recently made a few additions to our building codes and we've prepared a handy guide to any new construction you might be considering. In particular we've added a bunch of new new semantic elements that give you even more power to architect your pages. Now, our guide isn't exhaustive; rather, our goal here is to give you, the experienced builder, enough to be familiar with the new HTML5 elements and CSS3 properties so you can use them in the web applications you're learning how to build in this book when you're ready. So if you need a quick tutorial on the semantic additions to HTML5, take one—they're FREE (for a limited time only).



Webville Guide to HTML5 Semantic Elements

Here in Webville we've made some recent changes to our building code and we've prepared a handy guide for all your new construction. If you've been using <div>s for common construction like headers, navigation, footers, and blog articles, then we have some new building blocks for you. So make sure you're up to code.



<section>

A <section> is a "generic document". You could use <section> to mark up, oh, say a Guide to HTML. Or to enclose the HTML for a game. A <section> is *not* a generic container—that's <div>'s job. And remember, use <div> if you're just grouping elements together for styling purposes.

<article>

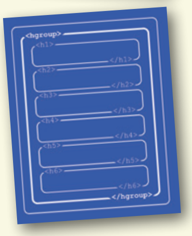
An <article> is a self-contained chunk of content that you might want to share with another page or web site (or even your dog). Perfect for blog posts and news articles.

<header>

<header> is for the tops of elements like <section> and <article>. You might also use <header> at the top of the body to create the main header for your page.

<footer>

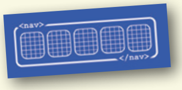
<footer> is for the bottoms of things. Things like <section>s, <article>s and <div>s. You might think you're only allowed one on a page; in fact, you can use it whenever you need footer content on a section of your page (like a bio or references for an article).



<hgroup>

This one can be tricky. Unlike <header>, which can contain any elements related to a header, <hgroup> is specifically for grouping headings (<h1>...<h6>) together inside a <header>. Good for outlines.

Webville Guide to HTML5 Semantic Elements



<nav>

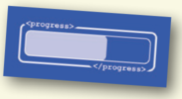
<nav> is navigation and for links, of course. But not just any links: use <nav> when you have a group of links, like navigation for your site, or a blogroll. Don't use it for single links in paragraphs.

<aside>

<aside> is handy for all kinds of things that are chunks of content outside the main flow of your page, like a sidebar, a pull quote, or an after-thought.

<time>

Finally! It's about time. You can mark up your times with <time>. No need to rush; take your time and do it right—you'll need to study up a bit on the valid formats for <time>.



<progress>

Almost done? Yes, we're making progress through these HTML5 elements...<progress> represents how far along you are in completing a task. Use with a little CSS and JavaScript for some nice effects.

<abbr>

Hey Mr., make sure you use an abbreviation for that long word! Great for search, because search engines aren't always as smart about abbreviations as we are.

<mark>

Use <mark> to mark words, for highlighting or editing, say. A good one to use with search engine results.

Adding style to your new construction with CSS3

Webville Guide to CSS3 Properties

Now that you've got your new building blocks into place, it's time to think about some interior design. You'll want to make all your new construction look good, right?

New properties

There are quite a few new properties in CSS3, many of which do what web page authors have been doing for years with various contortions of HTML, images, and JavaScript. Examples:

```
opacity: 0.5;
```

Makes an element 50% opaque

```
border-radius: 6px;
```

Creates a rounded effect with a 6px curvature on each corner

```
box-shadow: 5px 5px 10px #373737;
```

A shadow 5px long, 5px high, a blur of 10px and a dark grey color.

New layouts

There are a couple of powerful new ways to lay out your page with CSS that go beyond positioning and are much easier to use. Examples:

```
display: table;
display: table-cell;
```

This gives you a table layout without the HTML tables.

```
display: flexbox;
flex-order: 1;
```

With flexbox you have greater control over how the browser flows boxes, like <div>s onto the page.

New animations

With animations, you can animate between property values. For instance you can make something disappear by transitioning the opacity from opaque to translucent:

```
transition: opacity 0.5s ease-in-out;
```

The transition property specifies a property to transition into and out of, (in this case opacity), how long to take to do the transition and the easing function, so it's gradual.

```
opacity: 0;
```

By setting opacity to 0, say on a hover event, we can create a disappear/reappear animation.

New selectors

There are a whole slew of new selectors, including nth-child, which lets you target specific child elements enclosed in an element. Finally, you can set the background color of alternating rows in a list without going crazy.

```
ul li:nth-child(2n) { color: gray; }
```

This means: select every other list item and set the background color to gray.

Index

Symbols

A

B

C

D

E

F

K

L

M

T

X

Z

Colophon

All interior layouts were designed by Eric Freeman and Elisabeth Robson.

Kathy Sierra and Bert Bates created the look & feel of the Head First series. The book was produced using Adobe InDesign CS and Adobe Photoshop CS. The book was typeset using Uncle Stinky, Mister Frisky (you think we're kidding), Ann Satellite, Baskerville, Comic Sans, Myriad Pro, Skippy Sharp, Savoye LET, Jokerman LET, Courier New and Woodrow typefaces.

Interior design and production all happened exclusively on Apple Macintoshes—two Mac Pros and two MacBook Airs to be precise.

Writing locations included: Bainbridge Island, Washington; Portland, Oregon; Las Vegas, Nevada; Port of Ness, Scotland; Seaside, Florida; Lexington, Kentucky; Tucson, Arizona; and Anaheim, California. Long days of writing were powered by the caffeine fuel of Honest Tea, GT's Kombucha, and the sounds of Sia, Sigur Ros, Tom Waits, OMD, Phillip Glass, Muse, Eno, Krishna Das, Mike Oldfield, Audra Mae, Devo, Steve Roach, Beyman Brothers, Pogo, all the people at turntable.fm, and a heck of a lot more 80s music than you'd care to know about.

Don't you know about the web site? We've got answers to some of the questions in this book, guides to how to do more and daily updates on the blog from the authors!

This isn't goodbye

Bring your brain over to
wickedlysmart.com