

Remember, the first value is the horizontal offset of the shadow, the second value is the vertical offset, the third value is the blur, and the fourth value is the color. This will create outlined text like Figure 6-27.

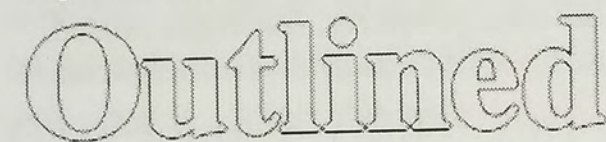


Figure 6-27. Outlined text created using the text-shadow property

■ **Note** WebKit-based browsers (Safari and Google Chrome) have implemented a property called text-stroke. It is not in any CSS specification at the W3C, so you're best to avoid it until (if) it appears in any of the CSS3 modules. To use it, you need the webkit prefix; the property is -webkit-text-stroke, which takes a width and a color, as in -webkit-text-stroke: 1px #f00.

Rules of typography

There's a rule of composition promoted in Robin William's book *The Non-Designer's Design Book* that states elements on the page that are merely similar should be avoided. If design elements on a layout are different, make them very different in order to create contrast in the layout. Use the contrasting differences among the design elements as a way to help visually organize the page. This fits into typography when you think about the typefaces you choose to use. Fonts fall into two main categories: serif and sans serif. Serif fonts are typefaces with embellishments to the edges of the letters (called **serifs**). These include Times New Roman, Georgia, Palatino, and so on. **Sans serif** (meaning "without serifs") includes Arial, Helvetica, Verdana, and so on. If you start including different serif or different sans serif fonts in the same design, pause for a moment and consider whether it is really necessary to have, for example, Arial and Helvetica together in the same layout. Typefaces that are similar yet distinct enough that their differences are noticeable are best avoided. Along those same lines, it's best to avoid using more than two typefaces overall on the page. There is so much you can do with bolding, italicizing, color, and so on, that even using one typeface offers a plethora of stylistic choices and treatments.

When thinking about the overall organization of textual content, use typography to create a hierarchy of information on the page, clearly identifying what content is more important to read over other content. This is the purpose of headlines, titles, subtitles, and so on; they create a hierarchy your eye can quickly scan to get deeper and deeper into the information on the page. Don't be afraid to size text large when you see it as the top bit of information on the page, but like with your typeface choices, avoid creating typography that is almost the same size but not quite the same, because this will likely be distracting.

The HTML5 outline algorithm discussed in Chapter 3 is particularly useful for helping you organize your page into a hierarchy. You may want to refer to that section when organizing your headings and so forth.

Lastly, the single greatest rule to follow in regard to typography is: if text is meant to be read, make it readable!

Summary

This has been one of the largest chapters in this book, and for good reason! I hope you can appreciate the power of CSS to fine-tune the appearance of elements on your page. This is the tip of the proverbial iceberg; there is plenty not covered that is developing in CSS, such as techniques for creating transitions and animations from CSS without any JavaScript code (the merits of which can be debated!), rotating and otherwise transforming the elements in 2D and 3D space, creating gradients from CSS, applying images to borders, and more! See www.w3.org/TR/css-2010/#properties for a list of available CSS properties you can use (because the W3 uses "snapshots," this may not be a complete list until it is updated again). We're not completely done with CSS, because this is a growing area, which, along with HTML5, is paving the path of what is possible on the road ahead, and that is where we will go in the next chapters!

User Interaction and the HTML5 APIs

Included in the HTML5 specification is documentation on how HTML elements, as well as the wider web browser environment, can be accessed from a programmed script. This is a component of the specification that is interwoven throughout the specification. Each component that can be scripted is broken into **application programming interfaces** (APIs) that define how the script can interface with specific elements on the page and specific aspects of the web browser. Some of these APIs are included as part of HTML5, such as the History API for manipulating the browser's forward and back buttons programmatically, while others are part of associated technologies and are covered in separate (but interlinked) specifications to HTML5, such as the geolocation API—which provides web pages with geographic location-aware capabilities. What these APIs are built around is the idea that the page and its environment are represented by a series of objects, such as the geolocation object, which contains location data for the page. In the context of just the page's content, these objects form the means by which a web page's Document Object Model (DOM) may be traversed, as discussed in Chapter 1.

The goal of this chapter is to get you versed in how to explore and use the scripting capabilities defined in the HTML specification so you have the necessary tools to explore on your own beyond what can be shown in this chapter. As examples, we will look at using scripting to interact with the browser's history, as well as the video and canvas elements. Lastly, we'll also look at how to add drag-and-drop support to any element.

Using JavaScript in this chapter

JavaScript has been sprinkled throughout the chapters in this book, but in this chapter we will be working with it more formally, so you'll want to set up a testing environment for JavaScript code. As discussed in Chapter 2, JavaScript can be embedded using the `script` element, or it can be loaded from an external file, which is the preferable method. Moving it to its own file creates a clearer separation between the markup and scripting on your website.

We're going to create a bare-bones web page to use as a template for the examples in this chapter. Create a directory named `jstemplate` (on your desktop or wherever is convenient for you to access). You'll use this directory as a starting template for the examples in this chapter. Launch your favorite text editor, and create the following web page:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>HTML5 Apprentice</title>
    <link rel="stylesheet" type="text/css" href="css/styles.css" />
    <script type="text/javascript" src="js/script.js" defer></script>
```



```

        </head>
        <body>
            <p>Content goes here!</p>
        </body>
    </html>

```

Save this code as a file named `index.html` in the `jstemplate` directory you created. Create a second page for the JavaScript named `script.js` and place it in a directory called `js` inside the `jstemplate` directory.

Write the following JavaScript in `script.js`:

```

// init function runs when page has fully loaded
function init() {
    // code to run
}
window.onload = init;

```

This script will run the function `init` when the page has fully loaded.

Lastly, create an empty CSS style sheet (just an empty text file at this stage) named `styles.css` and place it in a directory called `css` inside the `jstemplate` directory.

Accessing DOM properties and methods

Chapter 1 provided an overview of the DOM, which describes the elements on a page as a connected tree-like structure (technically an acyclic connected graph), which can be used to gain programmatic access to the page's elements, attributes, and text. What was also mentioned is that the DOM is accessible through a document object, which is contained inside another object—the window object. What wasn't mentioned is that more abstract entities (besides the web page document) can be accessed through the window object. For example, there is programmatic access available to the browsing history, website URL location, screen dimensions, and so forth. Each of these entities is accessible through the window as an object in its own right.

So, what exactly are objects? Objects are merely a collection of **properties** and **methods**. What does that mean? Properties are a series of keys and values, like attributes on HTML elements. They are what an object *has*. For instance, the width of the screen is a property that holds a specific value (the screen width in pixels). Some properties are read-only, while others can have their values overwritten. Properties are also known as variables, depending on the context.

Methods describe what a particular object can *do*. For instance, using JavaScript to pop up an alert window is done through a method. Methods are also known as **functions**. They are self-contained blocks of code that will run only when called, like the function you saw in the template JavaScript code in the previous section.

For instance, the window object has a property called `name` that can be used to add an identifier to a particular web browser window.¹ This property is not read-only, so it can be read and written to.

As far as methods go, the window object has a method (among many) called `alert()`, which you've likely seen in use. It pops up a dialog box with a message and an OK button. Using dot notation (described in Chapter 1), these could both be used like so:

¹ This may be used in the target attribute of hyperlinks in order to open a linked resource in a specific window.

```

function init() {
    // sets the name of the window
    window.name = "MAIN";
    // shows the name of the window
    window.alert("The name of this window is set to: " + window.name);
}
window.onload = init;

```

The specific usage of this property and method are beside the point at this stage; what is important is the conceptual difference between properties and methods and the knowledge that JavaScript can access settings and behavior in the web browser beyond just the HTML elements on the page.

Logging to the console

Chapter 1 also mentioned the web developer tools available to major web browsers and the code `console.log("message")`. Let's look at these two topics closer. Among the web developer tools in each browser, there will be a means to access a JavaScript console where messages from JavaScript can be logged. In the major web browsers, these are found as shown here:

- *Chrome*: Select View ► Developer ► JavaScript Console.
- *Firefox*: I recommend using the Console tab in Firebug.
- *Internet Explorer (8 and above)*: There is Script tab in the Internet Explorer Developer Tools, accessible by selecting Tools ► Developer Tools.
- *Opera*: Select View ► Developer Tools ► Error Console. Opera also has a toggle in Opera Dragonfly for showing and hiding a console.
- *Safari*: Select Develop ► Show Error Console.

If you modify your template script to log a message, you should see it appear in the JavaScript console when loading the page. Figure 7-1 shows an example in Safari of running the following code:

```

function init() {
    console.log("message");    // log a message to the JavaScript console
}
window.onload = init;

```

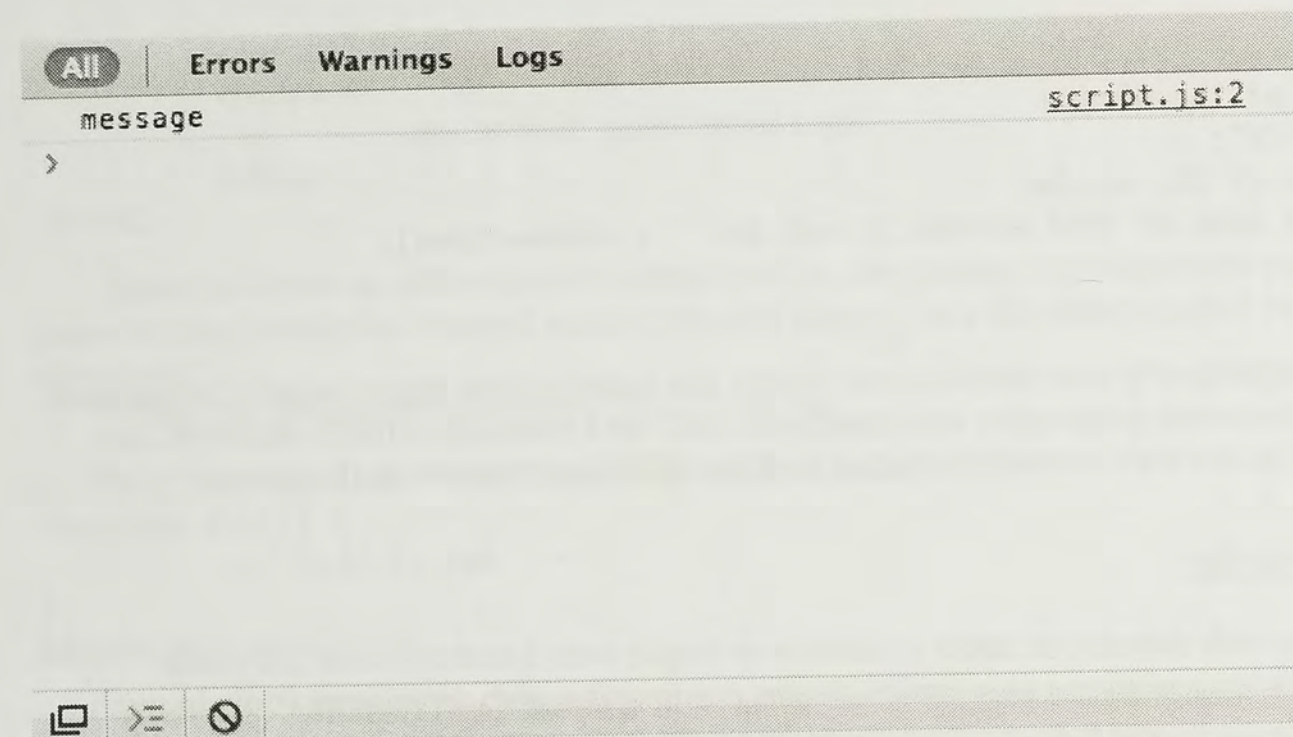



Figure 7-1. Safari's error console showing logged JavaScript output.

■ **Note** Another option for logging JavaScript output is Firebug Lite, a version of Firebug that can be used with Internet Explorer, Firefox, Opera, Safari, and Chrome. Download it here: <http://getfirebug.com/firebuglite>.

Although it is all well and good to log messages to the console, the real power of the console comes from logging parts of the DOM and inspecting the value and existence of objects, properties, and methods. For example, try changing the console message in the prior code snippet to the following, and reload the page:

```
... console.log(window);    // log the window object to the JavaScript console.
...
```

This will produce (depending on the JavaScript console) a representation of the window object as a collapsible list of properties and methods, as shown in Figure 7-2.

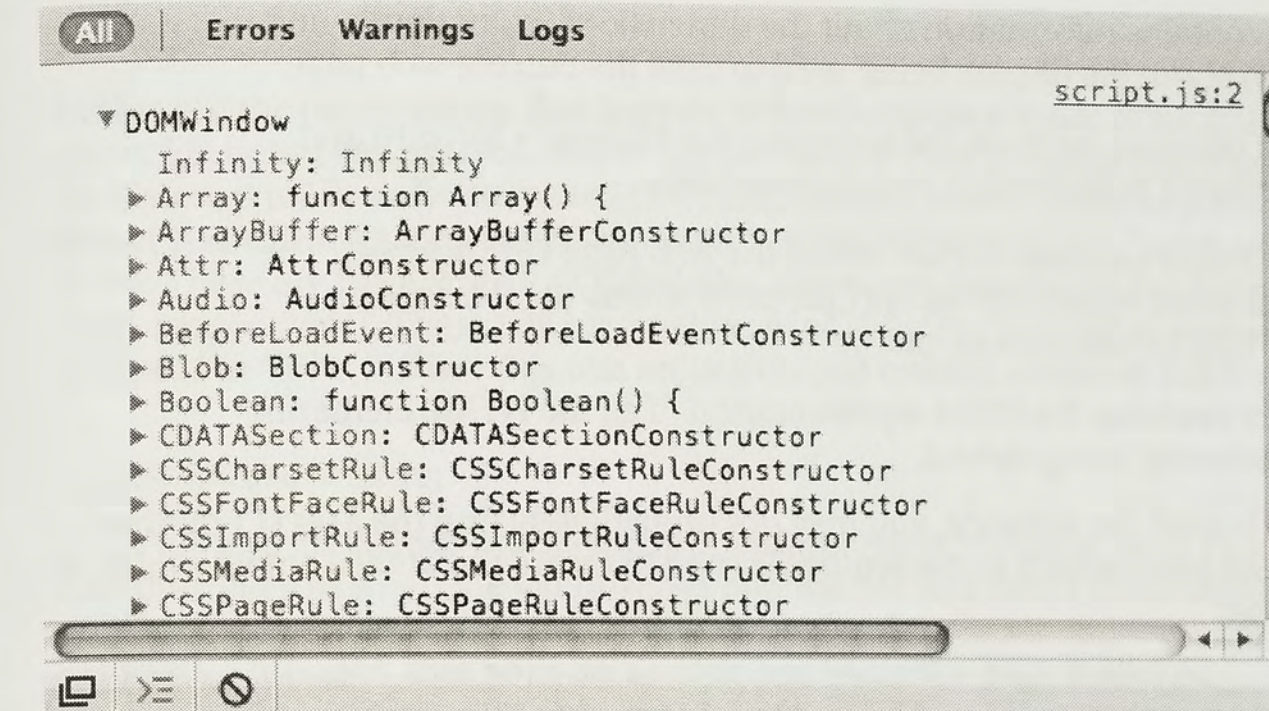


Figure 7-2. The window object as it appears logged in Safari's error console

Scrolling through this list, you will find interesting information about what you can manipulate from JavaScript. For example, you'll see the type of data you can work with from JavaScript, such as Arrays, Booleans, and so on. You'll also see a list of the HTML elements in the form of `HTMLAnchorElement`, `HTMLImageElement`, `HTMLParagraphElement`, and so forth. JavaScript is a prototype-oriented language, meaning it does not have classes (unlike Java or Adobe Flash's ActionScript) but instead utilizes "prototype" objects that provide a basis for other objects. These prototype objects can build on each other, creating a "prototype chain" that allows prototype objects to inherit properties and methods from other prototype objects. This process is analogous to the inheritance capabilities of classes in other object-oriented languages. The HTML element-like entries in the console output shown in Figure 7-2 are the prototype objects that define the properties and methods different HTML elements have when represented in the DOM. For instance, the `stop()` and `play()` methods defined for the video element that stop and play a video loaded in that element can be found through exploring the `HTMLVideoElement` object represented in the DOM (exploring this element, you'll discover it actually inherits these methods from `HTMLMediaElement`).

■ **Note** When logging HTML elements that appear on the page (by referencing them by their ID, for instance), the error console will normally show the actual HTML code for the element, which will look like a snippet of HTML source code. This behavior doesn't help you discover the properties that a referenced HTML element has, so in cases like this, use the console method `console.dir()` instead of `console.log()`. This method will show you the properties of a particular object sent to it.

Other objects of note in the window object include the following:

- The navigator object contains information about the web browser being used such as the plug-ins installed and vendor and version information. Also contained in this object is geolocation information.

- The screen object contains information about the dimensions, color depth, and related information about the display being used to view the current web page.
- The history object contains methods for accessing the browser's forward and back buttons and related functionality programmatically.
- The location object contains information about the web page URL address. The website can be redirected by setting the href property in this object, such as `location.href = "http://www.apress.com";`.
- The document object contains the DOM representation of all the HTML elements on the web page currently being viewed.

Some information will be buried; for instance, you may not immediately find the `alert()` method that was used earlier. For this we need to look at the prototype used to create the window object, which is accessible through the following:

```
...
console.log(window.constructor.prototype); // log the window object's prototype.
...
```

This will reveal the methods defined directly on the window object, as shown in Figure 7-3.

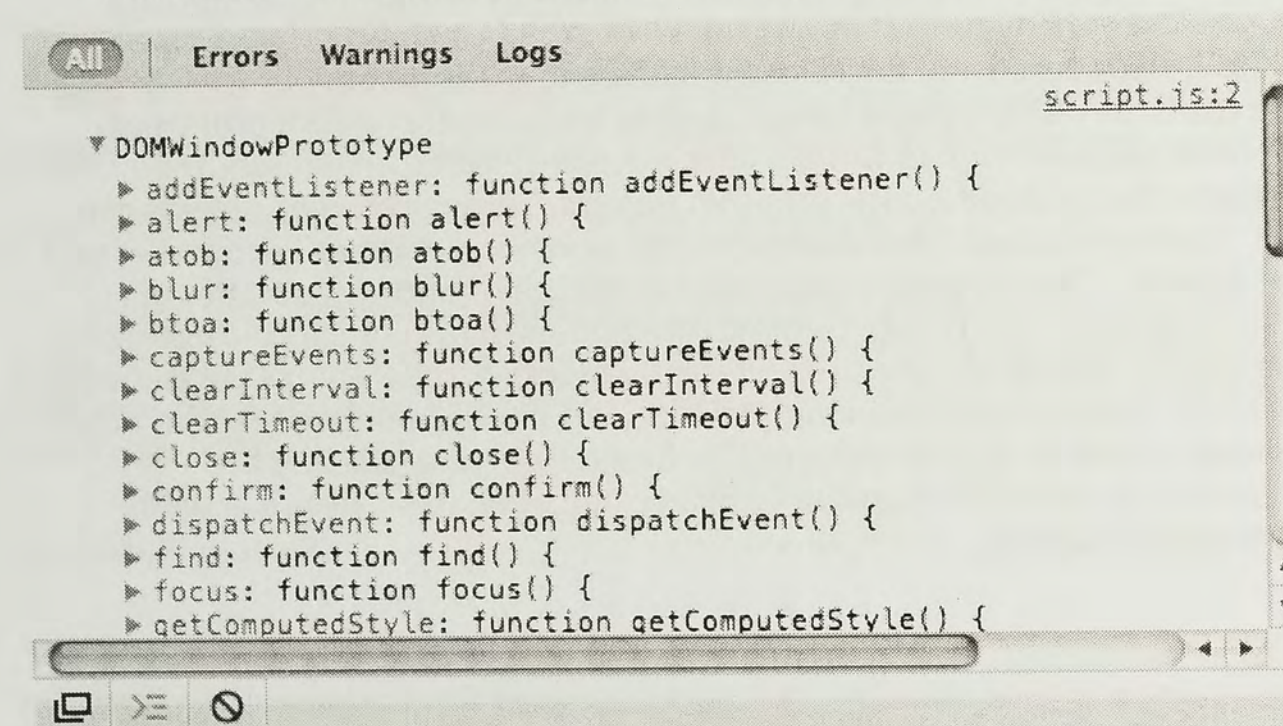


Figure 7-3. The window object's prototype object showing the methods defined for the window object

■ **Note** Accessing the object's prototype does not directly reveal the methods inherited by that prototype. For browsers that support it (it's nonstandard), such as Safari, Chrome, and Firefox, you will find a `__proto__` property that will give you access to the parent prototype that a particular prototype is built upon. This can be used to inspect the methods inherited from the parent prototype.

Events

Events are the notifications that happen in JavaScript as a result of such things as user input (mouse clicks, keyboard key presses, and so on) or conditions of the page (page has loaded, and so on) that can be “listened to” in order to make something happen when the event occurs. Elsewhere inside the window object you will see a long series of properties that begin with “on,” such as `onclick`, `onscroll`, `onload`, and so on. These are the events that the window object can respond to (clicking, scrolling, and loading the page, respectively). These will have values of null for the most part, except for `onload`, which will be set to function `init()`, since this was set in the code created earlier at this line:

```
...
window.onload = init;
```

This sets the `onload` property to our custom event handling `init()` function, meaning the `init()` function will run when the window has fully loaded and sent a notification of the `onload` event.

Each possible event can be associated with a function in this way. For example, the following would associate a function that pops up an alert box when the page is clicked:

```
function windowClickHandler() {
    alert( "Window was clicked" ); // pops up an alert box
}
window.onclick = windowClickHandler;
```

If the function associated with an event is given a parameter, then that parameter holds an object with information about the event that occurred. For instance, the following uses a parameter (named `e`) that contains a `pageX` and `pageY` property, which specifies the location of the mouse cursor on the page:

```
// logs the event object function windowMouseDownHandler(e) {
console.log(e);
// log the mouse location
console.log("Mouse is at: " + e.pageX + ", " + e.pageY);
window.onmousedown = windowMouseDownHandler;
```

When this code is run and the page is clicked, the event object (a `MouseEvent` object in this case) and the mouse location will be logged to the console.

The History API

Let's put all this JavaScript knowledge to use! I mentioned that the window object contains a history object for controlling the browser's forward and back buttons. This is actually not all that can be done with the history object, and HTML5 has built on the History API² to allow the actual history record to be updated programmatically. This means pages can be added to the history of the browser, and if the user actually clicks the back button, they will go to the pages that have been added to the history instead of pages they previously visited. The usefulness of this is evident in pages that use Ajax to load content dynamically. **Asynchronous JavaScript and XML (Ajax)** is a means of loading and injecting content into a page without having to reload the entire page. For instance, websites such as Twitter and Facebook use Ajax to add new posts to their pages.

² See www.w3.org/TR/html5/history.html.

Really simple Ajax

Ajax uses an object called XMLHttpRequest (XHR) to load content from a URL using JavaScript. This content then can be added to the page by using, for instance, the `innerHTML` property of HTML elements.

At its very simplest, using Ajax might look like the following:

```
var xmlhttp = new XMLHttpRequest();
xmlhttp.onreadystatechange = responseReady;
xmlhttp.open("GET", "content.txt", true);
xmlhttp.send();
```

The `var` keyword creates a new variable, which is just like a property—it's a container that holds some value that can be read or written to. The new variable in this case holds a new XMLHttpRequest object instance (the new keyword creates this object). A function is then set to handle the `onreadystatechange` event, which will be fired when the XHR object has retrieved some content. The `open()` method then specifies the kind of method used to retrieve the file, the filename, and whether the request is **asynchronous** (the script continues while the external file loads) or **synchronous** (the script waits for the external file to load before continuing). Setting this to `true` makes it asynchronous and `false` sets it to synchronous. The event handling function might look like this:

```
function responseReady() {
    document.body.innerHTML = xmlhttp.responseText;
}
```

This script will set the content on the page to the content that was inside the loaded text file.

As a complete example, we'll create a simple page with a next button and a previous button and a message that loads between the two. Duplicate the `jstemplate` directory from earlier and rename it `ajax`, and then edit `index.html` to look like the following HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>HTML5 Apprentice</title>
    <link rel="stylesheet" type="text/css" href="css/styles.css" />
    <script type="text/javascript" src="js/script.js"></script>
  </head>
  <body>
    <button id="prevButton">Previous</button>
    <p id="message"></p>
    <button id="nextButton">Next</button>
  </body>
</html>
```

This page contains two buttons and an empty paragraph between the two that we'll fill with content using Ajax. Open `script.js` from the `ajax/js` directory. Add the following script:

```
// create global variables
var xmlhttp;
var prevbutton;
var nextbutton;
var message;
var messageID = 1;
```

```
// initialize variables and add event listening functions
function init() {
    prevbutton = document.getElementById("prevButton");
    nextbutton = document.getElementById("nextButton");
    message = document.getElementById("message");

    prevbutton.onmousedown = prevButtonDown;
    nextbutton.onmousedown = nextButtonDown;

    checkButtons();
    loadFile();
}
// disable previous or next button depending on whether first or last message is displaying
function checkButtons() {
    if ( messageID == 1 ) prevbutton.disabled = true;
    else prevbutton.disabled = false;
    if ( messageID == 3 ) nextbutton.disabled = true;
    else nextbutton.disabled = false;
}
// decrement message ID when previous button is pressed
function prevButtonDown() {
    messageID--;
    if (messageID < 1) messageID = 1;

    checkButtons();
    loadFile();
}
// increment message ID when next button is pressed
function nextButtonDown() {
    messageID++;
    if (messageID > 3) messageID = 3;

    checkButtons();
    loadFile();
}
// load message files using Ajax
function loadFile() {
    var file = "message"+messageID+".txt";

    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = responseReady;
    xmlhttp.open("GET", file, true);
    xmlhttp.send();
}
// add Ajax loaded content to paragraph on the page
function responseReady() {
    message.innerHTML=xmlhttp.responseText;
}
window.onload = init;
```


This script uses a variable (`messageID`) to track which of three messages needs to load when the previous or next button is clicked. You'll need to create three text files to load named `message1.txt`, `message2.txt`, and `message3.txt` and save them in the `ajax` directory. Add some text to each of these files; just make sure the text is different for each. This text will be injected into the empty paragraph element on the page.

Open `index.html` in a web browser to test the file. You should be able to click the buttons back and forth and display a different message on each click (Figure 7-4).

Previous

Message 3

Next

Figure 7-4. The output of a simple Ajax application after two clicks of the next button. The button is disabled when the last of the three messages is displayed.

History-enabled Ajax

If you look at the URL for the page's address in the previous example, you will see it is not updating as the different content is loaded. This isn't ideal because the pages showing the different messages can't be bookmarked, and clicking the browser's back button will not go back to the previously displayed message but instead will go to the previous page viewed. However, the History API can be used to overcome these issues. HTML5 introduces two methods, `pushState()` and `replaceState()`, for adding to and editing the browsing history. The first, `pushState()`, allows a new entry to be added to the page's browsing history, while `replaceState()` will replace the current browsing location in the history with a new entry.

Let's edit the script in the previous example and include `pushState()` to add a new entry to the browsing history when the next and previous buttons are clicked. Go back to the previous script and edit the `prevButtonDown()` and `nextButtonDown()` methods:

```
...
function prevButtonDown() {
    messageID--;
    if (messageID < 1) messageID = 1;

    var obj = { page: messageID };
    var title = "page"+messageID;
    var url = "#message"+messageID;
    window.history.pushState( obj , title , url );

    checkButtons();
    loadFile();
}
function nextButtonDown() {
    messageID++;
    if (messageID > 3) messageID = 3;

    var obj = { page: messageID };
```

```
var title = "page"+messageID;
var url = "#message"+messageID;
window.history.pushState( obj , title , url );
```

```
checkButtons();
loadFile();
```

```
}
...
```

The `pushState()` method takes three arguments. The first is a JavaScript object (the curly brackets are shorthand notation for creating an object) that can contain information about the page being added to the history. In this case, it's an object with a single custom property, `page`, and a value that holds the page ID. Next is the title of the page added to the history (for example, "page1"). Lastly—and most importantly—is the URL of the page to add to the history. The page we'll add is the same page we're on, but we'll add a hashtag to the URL so it becomes `ajax.html#message1`, `ajax.html#message2`, and `ajax.html#message3` for each click of the next button. We can just add the hashtag without specifying the current page, because the browser will add the current page if no page is specified.

OK, edit the `loadFile()` function so that it retrieves the message ID from the hashtag in the URL:

```
...
function loadFile() {
    // retrieve the hashtag from the URL using the location object
    var messageHash = window.location.hash;
    if (messageHash == "") messageHash = "#message1";
    var file = messageHash.substr(1)+".txt"; // strip out the "#" from the hashtag

    xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = responseReady;
    xmlhttp.open("GET",file,false);
    xmlhttp.send();
}
...
```

Now test the page and use the next and previous buttons. You should see the web page address URL update with the hashtag. This means the individual Ajax-enabled pages of content can be bookmarked, which is great! However, if you click the browser's forward or back button, you will notice the message on the page does not update, even though the URL in the address bar does. The reason for this is that the `onload` event does not fire again when the user clicks the forward and back buttons. This is because the page has been cached by the browser so that it will load faster as the user navigates through the history. What we need to add is an event that fires when the history has changed, and thankfully there is such an event called `onpopstate`. At the very end of the script, add the following:

```
...
window.onpopstate = init;
```

This will run the `init()` function every time the history is navigated. This isn't the most efficient code, because there will be redundant calls to the `init()` function (for example, twice when first loading the page), but use this simple example to develop from.

Building a custom video controller

Let's look at interacting with the JavaScript methods associated with an HTML element. Earlier I mentioned that the video element has a `play()` method and a `stop()` method. We can use these to control video playback from JavaScript, which means we can create a control bar with a custom play and pause button. Say goodbye to a control bar that looks different in every browser! For the graphics we'll use CSS, but more specifically we'll use a concept called **CSS sprites**, which is a technique for using one image file to display two or more images at different points on the page. The idea behind this technique is that a single image file can contain a whole bunch of images, which then can be cropped and positioned in different ways to show the different images. This is kind of like looking through a paper towel tube at cookies spread out on a cookie sheet. You'll be able to see only one cookie at a time, but you can move the sheet around to show different cookies. This technique gives the illusion of multiple images having been downloaded, but in fact only one has, meaning all the images will appear on the page at once and only one request has gone to the server to fetch the image.

On to our example...duplicate the `jstemplate` directory from earlier and rename it `video_player`. Create an image for the sprite that contains a play and pause graphic for our video controls (Figure 7-5). Set the size to 80 pixels wide by 30 pixels tall.



Figure 7-5. An image sprite with two images for the controls on a simple video player

Update the HTML in `index.html` to look like the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>HTML5 Apprentice</title>
    <link rel="stylesheet" href="css/styles.css" type="text/css" />
    <script type="text/javascript" src="js/script.js" defer></script>
  </head>
  <body>
    <section>
      <video width="320" height="240" poster="poster.jpg" id="my_video">
        <source src="trailer.webm" type='video/webm; codecs="vp8, vorbis"' />
        <source src="trailer.mp4" type='video/mp4; codecs="avc1.4D401E, mp4a.40.2"' />
        <source src="trailer.ogv" type='video/ogg; codecs="theora, vorbis"' />
        <p>Video playback not supported.</p>
      </video>
      <div class="video_controls">
        <a href="nojs-player.html" title="Play" id="control_button">
          <span id="control_graphic" class="play"></span>
          <label id="control_label">Play</label>
        </a>
      </div>
    </section>
  </body>
</html>
```

As you can see, this code contains the video element with fallback content³ followed by a `div` area for the custom video control. To get this example to work, you will have to include three video files sized to 320 by 240 pixels, which are named `trailer.webm`, `trailer.mp4`, and `trailer.ogv` in the same directory as `index.html` (it wouldn't be a bad idea to create a video directory or similar for the videos, if you do this be sure to update the HTML).

Next we'll create the JavaScript that will be used to enable this custom control. Open `script.js` from the `video_player/js` directory. For the code, first we will create variables to hold a reference to the video, anchor (`<a>`), `span`, and `label` on the page. Next the `init()` function is updated so that it sets the value of each JavaScript variable to a reference to an HTML element, which can be retrieved through their `id` attribute using the `getElementById()` method. Update `script.js` to look like this:

```
// JavaScript variables used to hold references to HTML elements
var video;
var control_button;
var control_label;
var control_graphic;

// initialization function runs when the page has loaded
function init() {
  video = document.getElementById("my_video");

  control_button = document.getElementById("control_button");
  control_label = document.getElementById("control_label");
  control_graphic = document.getElementById("control_graphic");

  control_button.onclick = playVideo;
}

window.onload = init; // runs the init function when the page has finished loading
```

■ **Note** Classes instead of IDs could be used for all the elements, and the `document.getElementsByClassName()` function could have been used to automatically enable functionality for multiple video controls on one page, but this would make the code a bit more complex, so for brevity's sake IDs have been used.

The last line in the `init()` function calls another function when the button control is clicked. This function is named `playVideo()`. Go ahead and add it to `script.js`:

```
...
function playVideo() {
  control_graphic.className = "pause";
  control_button.onclick = pauseVideo;
  control_button.title = "Pause";
}
```

³ For brevity, the fallback content has been shortened; refer to Chapter 5 for comprehensive fallback content information for the video element.


```

        control_label.textContent = "Pause";

        video.play();

        return false;
    }

```

This function sets the class attribute on the control graphic to a CSS class named `pause`, which positions the CSS sprite over the pause button image. Next, it sets the text content (which is used as fallback content if the CSS style sheet was disabled), and it then plays the video. The `return false` bit ensures the link (with the ID of `control_button`) does not go to its linked page when it's clicked; however, if JavaScript is disabled, it'll go to a page called `nojs-player.html` that would presumably contain a video player that will work without JavaScript (you'll have to build this one on your own). This function also sets a new function for when the button control is clicked again. This new function is called `pauseVideo()`, and it pauses the video and reverses the changes set in the `playVideo()` function. Go ahead and add this to your `script.js` file:

```

function pauseVideo() {
    control_graphic.className = "play";
    control_button.onclick = playVideo;
    control_button.title = "Play";
    control_label.textContent = "Play";

    video.pause();

    return false;
}

```

That's all the JavaScript we need; now on to the CSS. First the link anchor element is set to display as a block-level element and has its width and height set. The property `overflow` is set to `hidden` so that content that flows outside of the dimensions of the controls does not show up. This is needed to hide the fallback text, because it will be pushed out of the bounds of the control. Open `styles.css` from the `video_player/css` directory. Add the following CSS rule:

```

.video_controls a {
    display: block;
    width: 40px;
    height: 40px;
    overflow: hidden;
}

```

Next we add a rule for the span so that it is styled to be a block-level element as well (pushing the other textual content out of the way). This is where the background image sprite (named "controls.png") is attached:

```

.video_controls span {
    display: block;
    background-image: url(controls.png);
    background-repeat: no-repeat;
    width: 40px;
    height: 40px;
    background-color: #ccc;
}

```

```

}

```

Lastly two CSS classes are created that set the position of the background image, moving it to show the play symbol or pause symbol:

```

...
.pause {
    background-position: -45px 5px;
}
.play {
    background-position: 5px 5px;
}

```

That's it! The play and pause buttons are derived from the same image, as shown in Figure 7-6.



Figure 7-6. The play (L) and pause (R) states of the custom video control using a CSS sprite

Scripted 2D Canvas API

Now let's look at something even more interactive. As you saw in Chapter 5, the canvas element does not have much that can be done to it purely with HTML, because it has only two attributes, which simply specify the width and height of the canvas area. The real power of canvas comes from manipulating the element from JavaScript. Remember, the canvas is a bitmap canvas, meaning it is essentially a blank image that we can manipulate the pixels of through a set of drawing commands.

To work with the canvas, a particular "context" needs to be retrieved from the element, which then can be used as the target of drawing operations. The context specifies what kind of image we are dealing with. There are two choices available: a two-dimensional or three-dimensional image.

Let's begin by hunting down the method for retrieving the context we will draw with.⁴ First duplicate the `jstemplate` directory from the beginning of the chapter and rename it `canvas`. Modify the HTML in `index.html` to add a canvas element (and some fallback content) in place of the paragraph of text:

⁴ For the impatient, the method for retrieving a context is called `getContext()`.


```

...
<body>
  <canvas id="canvas" width="300" height="300">
    <p>The canvas element is not supported!</p>
  </canvas>
</body>

```

Now modify `script.js` in the `canvas/js` directory to gain access to the canvas element from JavaScript. We'll begin by adding a variable that will hold a reference to the canvas element, which we'll set using the `getElementById()` method:

```

var canvas;           // variable to hold a reference to the canvas element on the page

function init() {
  canvas = document.getElementById( "canvas" );      // look up the canvas element by its ID
}
window.onload = init;

```

■ **Note** You may have noticed earlier that `document` is a property of the window object, so why isn't the syntax `window.document...` in the previous code block? The reason is that the browser will automatically look into the window object when looking up a property or method, so it isn't necessary to prefix all the properties or methods of the window object with `window`. For example, `console` is also a property of `window`, but for brevity `window` is usually left out (although it could be included with no harm caused).

This gets us access to the canvas element from our script. Let's look at what methods are defined for the canvas element by looking at its prototype, like we did with the window object, by accessing `constructor.prototype`. Edit the script like so:

```

...
function init() {
  // look up the canvas element by its ID
  canvas = document.getElementById( "canvas" );
  // log the canvas element's prototype
  console.log( canvas.constructor.prototype );
}
...

```

This reveals `getContext()`, which is the method we need to retrieve the context! The other method, `toDataURL()`, is for converting the canvas image data to a URL that could, for instance, be supplied as the source image data for an image element (`img`).

As mentioned earlier, the context will tell what type of image we are dealing with, which can be a two-dimensional or three-dimensional image. To retrieve a certain context, a text keyword is given as an

argument to the `getContext()` method.⁵ To retrieve a two-dimensional context, the text keyword "2d" is used. To retrieve a three-dimensional context, the text keyword "webgl" is used instead. As you can see, the 3D context uses WebGL for interacting with the image.

■ **Note** WebGL is a 3D rendering specification for web browsers that is overseen by the Khronos group.⁶ WebGL is an experimental feature in web browsers, and as such, the `webgl` keyword may not work for the immediate future. If you experiment with this context and find it does not work, try "experimental"-webgl instead, which is the temporary context keyword being used in supporting browsers until WebGL is further along in its development.

Let's retrieve the 2D context and inspect its properties and methods using the JavaScript console; edit the script to look like the following:

```

var canvas;
var context;           // variable to hold a reference to the canvas context

function init() {
  canvas = document.getElementById("canvas");
  // retrieve the 2D canvas context
  context = canvas.getContext("2d");

  // inspect the canvas context
  console.log(context);
  // inspect the canvas context prototype
  console.log(context.constructor.prototype);
}
window.onload = init;

```

This will log two objects to the console, `CanvasRenderingContext2D` and `CanvasRenderingContext2DPrototype` (the actual names may vary by browser). The first will show the properties available on the context, which includes the color of lines and fills added to the canvas. The `CanvasRenderingContext2DPrototype` shows the available methods, which includes a range of methods for drawing on and transforming the canvas. The graphical capabilities of the canvas encompass the following:

- *Basic drawing:* There are methods for drawing rectangles, lines, curves, and arcs.
- *Fills and strokes:* There are methods for creating solid fills and outlines.
- *Effects:* There are methods for creating shadows, gradients, transparency, and for compositing images on top of each other.

⁵ The WHATWG runs a wiki at <http://wiki.whatwg.org/wiki/CanvasContexts> that provides an overview of the canvas context keywords available.

⁶ The WebGL specification is available here: www.khronos.org/registry/webgl/specs/latest/.

- *Transformations:* There are methods for scaling, rotating, and translating (moving) the image.
- *Text:* There are methods for adding solid or outlined text.
- *Images:* There are methods for drawing an image (or even a video or another canvas element) onto the canvas, which can then be transformed or otherwise manipulated.

We will explore the first two areas, but be sure to use the console to explore the methods and properties available that are not covered in this chapter. The purpose of many of them is quite self-evident!

Drawing on Canvas

Let's begin by drawing a rectangle on the canvas that covers the available area. The methods related to drawing a rectangle are as follows:

- `fillRect(x,y,w,h)`: Draws a solid rectangle
- `strokeRect(x,y,w,h)`: Outlines a rectangle

Using either of these methods, we can create a rectangle of a given width and height starting at a specified location on the canvas. Before using these methods, you may want to style the rectangle to specify the appearance of the fill and stroke (the outline color). There are a few properties that can be set for this purpose:

- `fillStyle`: The fill color
- `strokeStyle`: The color of the outline
- `lineWidth`: The width of the outline

To set the style properties, set the value equal to a quoted CSS-style color code, such as `#000000` for black. The `lineWidth` property takes a number that specifies the width in pixels. To fill the entire canvas with a solid rectangle, you'd specify a starting point of 0, 0 (the upper-left point) and a width and height the same as the canvas width and height. Adding an outlined rectangle on top of that would give us the following:

```
...
canvas = document.getElementById( "canvas" );
context = canvas.getContext( "2d" );
// the color of the fill
context.fillStyle = "#cccccc";
// the color of the outline
context.strokeStyle = "#999999";
// the width of the outline
context.lineWidth = 5;
// fill the canvas area with a rectangle
context.fillRect( 0, 0, canvas.width, canvas.height );
// outline a rectangle inside the canvas borders
context.strokeRect( 30, 30, 200, 100 );
...
```

Add this code to the content of the `init()` function in `script.js` in the `canvas/js` directory to see it in action! The canvas uses a Cartesian coordinate system with an inverted y-coordinate. This means every pixel on the canvas can be specified by an x and y value, where x is the number of pixels from the left side of the canvas area and y is the number of pixels from the top side of the canvas area. For instance, an x, y coordinate of 30, 30, would specify a location 30 pixels from the left side and 30 pixels from the top side of the canvas area (Figure 7-7).

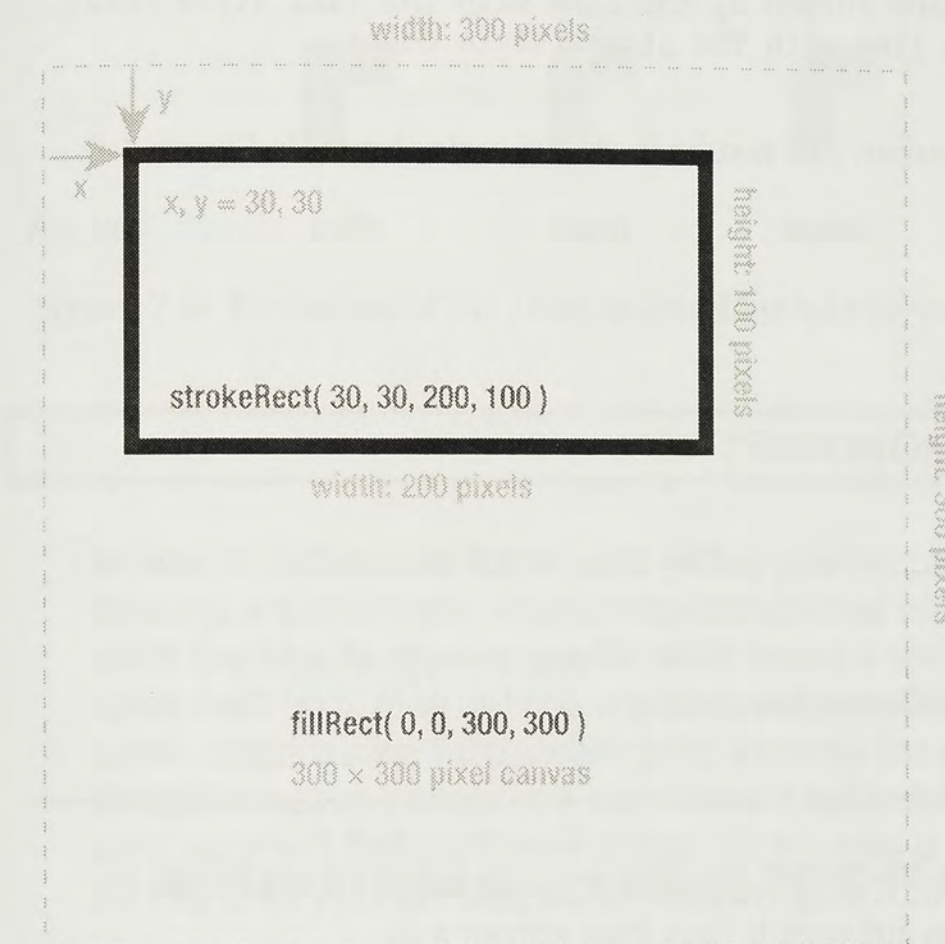


Figure 7-7. The rectangle drawn by `strokeRect(30,30,200,100)` in a 300 by 300 pixel canvas

Drawing more complex shapes than a rectangle is more complicated. A line needs to be created—also known as a **path**—which can be used to build the shape segment by segment. The following are the relevant methods for drawing a simple shape with straight edges:

- `beginPath()`: Starts a new line
- `closePath()`: Finishes a new line
- `moveTo(x,y)`: Moves to a coordinate on the canvas
- `lineTo(x,y)`: Draws a line segment to a coordinate on the canvas
- `stroke()`: Colors the line
- `fill()`: Fills in the shape created by the line segments

The `beginPath()` method is first used to tell the canvas that a new shape is being drawn, which consists of line segments. The `moveTo()` and `lineTo()` methods are then used to move around the canvas and draw line segments. Lastly, the `closePath()` method is used to complete a line and connect the beginning and ending points if the line is filled. For example, to create a triangle a path is begun, the drawing point is moved, two lines are drawn, and the path is closed, filled, and outlined:


```

...
context.beginPath();           // start a new line
context.moveTo(50,50);         // move to 50 pixels from the left and top edge of the canvas
context.lineTo(150,50);        // draw a line to 150 and 50 pixels from the left and top edge
context.lineTo(100,150);       // draw a line to 100 and 150 pixels from the left and top edge
context.closePath();           // close the line
context.fill();                // fill the shape formed by the line with the fill style color
context.stroke();              // outline the line with the stroke style color
...

```

Used in place of the rectangle drawing code earlier, this results in the triangle shown in Figure 7-8.

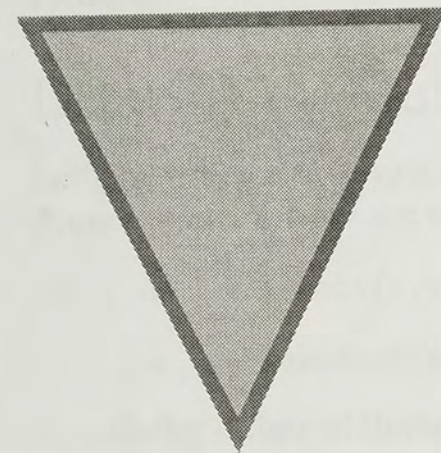


Figure 7-8. A triangle drawn using the canvas path drawing methods

■ **Note** If the `beginPath()` and `closePath()` methods are omitted, the lines won't be automatically closed to form a shape, which may be desired in cases where you just want to have lines without a fill.

In place of the `lineTo()` method, the `arcTo()`, `bezierCurveTo()`, or `quadraticCurveTo()` methods could be used to create shapes with curved edges.

Additionally, the `lineCap` and `lineJoin` properties can be set to affect how the end of line segments and the join points of each segment appear. The `lineCap` property can be set to `butt`, `round`, or `square` to change the end of a line. The `lineJoin` can be set to `miter`, `round`, or `bevel` to affect the shape of the join point of two line segments (Figure 7-9). When set to `miter`, another property, `miterLimit`, can be used to specify the angle over which a point will be formed.

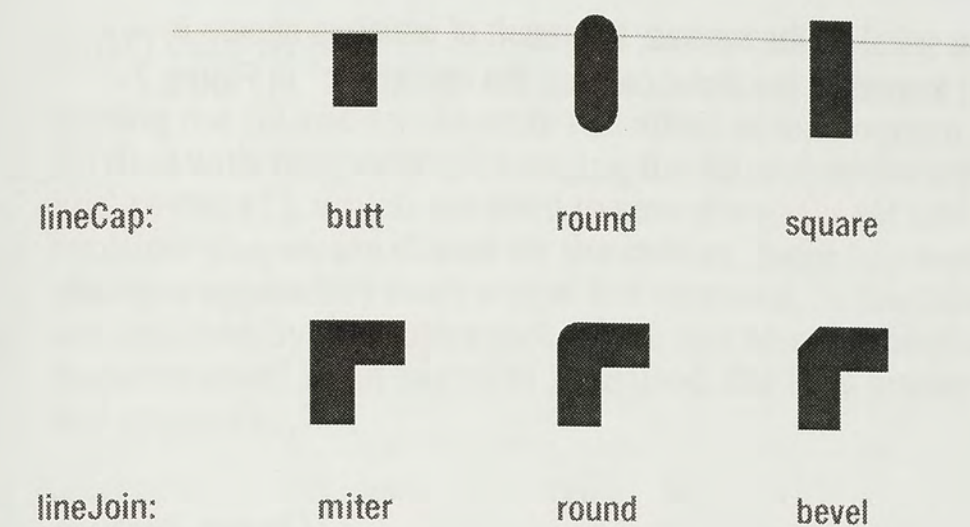


Figure 7-9. The values of the `lineCap` and `lineJoin` properties

TRIGONOMETRY

An area of mathematics that is worth getting your head around when dealing with algorithmically created drawings is trigonometry. Trigonometry is concerned with the study of the angles and sides of triangles, which is critical for drawing specific points around a circle. Since the canvas uses a Cartesian coordinate space made up of rows and columns (the *x* and *y* coordinates), finding the *x*, *y* coordinates of a point a certain distance away from another point, when the line of sight between them is slanted, is not as straightforward as it would be if they were in a horizontal or vertical line from each other, but finding the point really isn't that much more difficult. All you need is the angle formed inside a triangle drawn between the two. Once you have that, the formulas in Figure 7-10 are all you need.

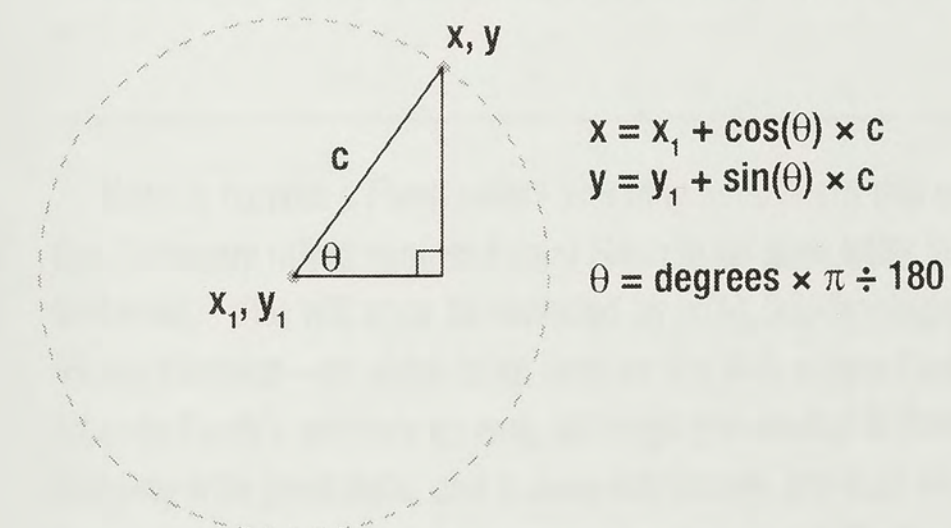


Figure 7-10. Determining the point on a circle using trigonometric functions

In Figure 7-10, the angle, *theta*, needs to be in radians, which equals the angle in degrees times *pi* divided by 180. Translated into code, the formula in Figure 7-10 might look like this:

```

x = 50 + Math.cos( 45 * Math.PI / 180 ) * 20;
y = 50 + Math.sin( 45 * Math.PI / 180 ) * 20;

```

This would find the *x* and *y* coordinates of a point that was at a 45-degree angle and was 20 pixels out from another point at the *x*, *y* coordinate of 50, 50 on the canvas.

Let's look at a full example. This example draws a spiral on the canvas, the result of which is shown in Figure 7-11. It runs a loop a number of times that increases the distance from the center ("c" in Figure 7-10) and increases the angle in degrees:

```
// declare the global variables
var canvas;
var context;
var centerX;
var centerY;
var degrees = 0;
var radius = 1;

function init() {
    canvas = document.getElementById( "canvas" );
    context = canvas.getContext("2d");
    centerX = canvas.width/2;
    centerY = canvas.height/2;
    // move to the center of the canvas
    context.moveTo( centerX , centerY );
    // loop two thousand times, drawing a line out further and further from the center
    for (var i=0; i<2000;i++) {
        degrees += 1;
        radius += 0.02;
        context.lineTo(
            centerX+Math.cos(degrees*Math.PI/180)*radius ,
            centerY+Math.sin(degrees*Math.PI/180)*radius
        );
    }
    context.stroke();
}
window.onload = init;
```

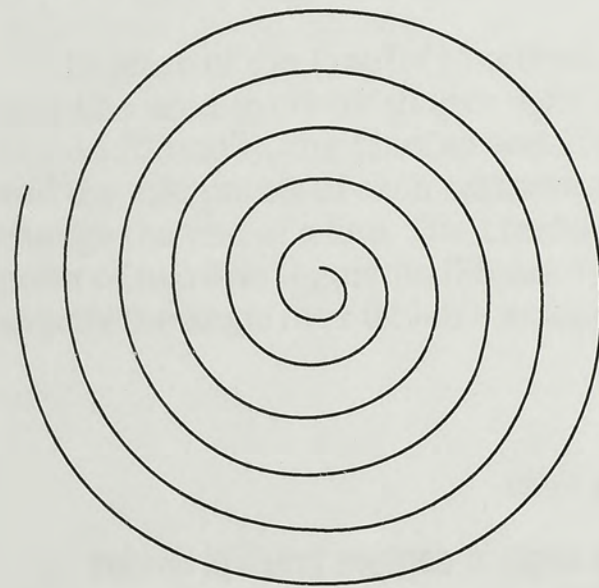


Figure 7-11. Spiral drawn on canvas using trigonometric functions

Canvas state

Setting the fill and stroke style will affect all subsequent graphics created on the current canvas context. To deal with temporarily changing the fill and stroke style, the canvas context has two methods, `save()` and `restore()`, which are used to save the styles set and then restore them to the saved state later. When multiple shapes are drawn on the canvas, these two methods are often used to isolate the stylistic changes applied by each shape. For instance, in the following code block, three squares are drawn that are outlined by the colors red, green, and blue, respectively. A border is then drawn around all the boxes. Because `save()` and `restore()` are used, the final border does not need a style set because it will be using the original styles:

```
...
context.save();
context.strokeStyle = "#ff0000";
context.strokeRect(0,0,100,100);
context.restore();

context.save();
context.strokeStyle = "#00ff00";
context.strokeRect(50,50,100,100);
context.restore();

context.save();
context.strokeStyle = "#0000ff";
context.strokeRect(100,100,100,100);
context.restore();

// Rectangular outline will be default black, because original style settings were restored
context.strokeRect(0,0,200,200);
...
```

■ **Note** Is canvas a Flash killer? You may have heard this question posed, particularly after Steve Jobs of Apple Inc. famously railed against Adobe Flash in an open letter in 2010, in which he stated that Flash is an antiquated technology that will soon be replaced by HTML5 technologies. This was in part spurred on by the creation of the video element—as video is an area on the Web where Flash dominates—but the canvas element encroaches a bit onto Flash's territory as well, although the overlap is limited. Canvas is a scriptable bitmap image optimized for drawing with pixel data, and it does not include any built-in methods for animation. Canvas uses immediate mode rendering, meaning it does not store the graphics it displays as separate entities. It's like actually painting on a real canvas, where paint can be layered on top of the image, but once on the canvas, it is effectively part of any prior paint that has been applied. Flash content, on the other hand, is traditionally stored as vector imagery (although it handles bitmap data as well), and it includes the concept of a display list for its graphics, which is very much like the DOM for HTML. This means that graphics in Flash are separated into nodes that can be easily accessed from code for interactive and animation purposes. In many respects, SVG, another graphics technology for the Web, is more akin to the traditional space of Flash. Therefore, canvas alone does not replace Flash, but

together with other technologies associated with HTML5, it is making the authoring of content traditionally restricted to Flash possible.

Canvas interactivity

Interacting with canvas graphics is a matter of setting an event handler function on one of the mouse-related events of the canvas, such as `onmousedown`, `onmouseover`, or `onmousemove` (use `console.log()` on the window object to explore the events available) and then doing some drawing on the canvas at the mouse location. For example, the following script records the x, y coordinate of the mouse cursor position and draws a line from the saved to the current position every time the user moves the mouse, creating a simple drawing application (Figure 7-12):

```
// declare global variables
var canvas;           // reference to canvas element on page
var context;          // reference to canvas context
var cwidth;           // reference to canvas width
var cheight;          // reference to canvas height
var lastX = 0;        // variable to hold an x coordinate value
var lastY = 0;        // variable to hold an x coordinate value

// initialize the variables and add event handler
function init() {
    canvas = document.getElementById( "canvas" );
    context = canvas.getContext("2d");
    cwidth = canvas.width;
    cheight = canvas.height;
    context.strokeStyle = "#000000";
    context.strokeRect(0,0,cwidth,cheight);

    // call the draw function when the cursor moves over the canvas
    canvas.onmousemove = draw;
}

// draw on the canvas
function draw(e) {
    // update the saved x, y coordinates to the position of the cursor
    // if it is first entering the canvas
    if (lastX == 0) lastX = e.pageX - canvas.offsetLeft;
    if (lastY == 0) lastY = e.pageY - canvas.offsetTop;
    // begin a new line and move to the last saved x, y coordinates
    context.beginPath();
    context.moveTo(lastX, lastY);
    // set the saved x, y coordinates to the position of the mouse cursor
    lastX = e.pageX - canvas.offsetLeft;
    lastY = e.pageY - canvas.offsetTop;

    // draw a line
    context.lineTo(lastX, lastY);
    context.closePath();
    context.stroke();
}
```

```
}
window.onload = init;
```



Figure 7-12. Doodle made with a simple canvas drawing application

Canvas animation

Animation is just a series of still images, so the basic steps involved in animating are as follows:

1. Clear any existing graphics on the canvas.
2. Draw new graphics on the canvas.
3. Repeat steps 1 and 2

Drawing on canvas uses the drawing methods covered earlier, but clearing the canvas introduces a new method: `clearRect(x,y,w,h)`. This method clears all graphics from the canvas within a rectangular area. For animations, this would usually mean clearing the entire canvas area, unless you could reliably know only one portion of the canvas would be changing throughout the animation.

The global window object contains two methods for repeating a piece of code over time, as is needed in an animation: `setInterval(f,t)` and `setTimeout(f,t)`. Both of these methods call a custom function (f) after a specified interval (t). The interval is set in milliseconds, so 1000 would be one second in length. The difference between the two is `setInterval(f,t)` runs continuously, while `setTimeout(f,t)` executes only once after the set amount of time. We'll use `setInterval(f,t)` to create a basic animation engine.

■ **Note** Firefox, Chrome, Opera, and Internet Explorer have an experimental method called `requestAnimationFrame()`, which is used to tell the web browser to repaint the window in order to redraw the frames of an animation. This has advantages over using the `setInterval()` method because the animation runs only when the browser is available, meaning that, for instance, an animation running in a tab that is hidden will stop because the page is not being shown, saving the computer's processor from consuming resources unnecessarily. You can find more information on this method at <https://developer.mozilla.org/en/DOM/window.requestAnimationFrame>.

To create a basic animation engine, the following functions are created:

- `init()` to set up initial variables and start the animation
- `invalidate()` to specify that the animation frame needs to be redrawn
- `clear()` to clear the animation frame
- `draw()` to draw the animation frame
- `update()` to update variables used when drawing the animation frame

Put together, the code for a basic animation engine looks like the following, which will animate two lines across the canvas (Figure 7-13):

```
// declare global variables
var canvas;
var context;
var cwidth;
var cheight;
var lastX = 0;
var lastY = 0;

// initialize animation
function init() {
    // set variable values
    canvas = document.getElementById( "canvas" );
    context = canvas.getContext("2d");
    cwidth = canvas.width;
    cheight = canvas.height;

    // start animation sequence by calling the update function every 30 milliseconds
    setInterval(update, 30);
}

// clear and redraw the canvas
function invalidate() {
    clear();
    draw();
}

// clear the canvas
function clear() {
    context.clearRect( 0 , 0 , cwidth , cheight );
}

// draw the graphics on the canvas using the saved x, y coordinate values
function draw() {
    context.fillRect(0,lastY,cwidth,5);
    context.fillRect(lastX,0,5,cheight);
}

// update the saved x, y coordinate values. Reset them to zero if they reach the canvas edge
function update() {
    lastY += 1;
    if (lastY > cheight) lastY = 0;

    lastX += 1;
    if (lastX > cwidth) lastX = 0;
```

```
        invalidate(); // call invalidate to redraw the canvas
    }
    window.onload = init;
```

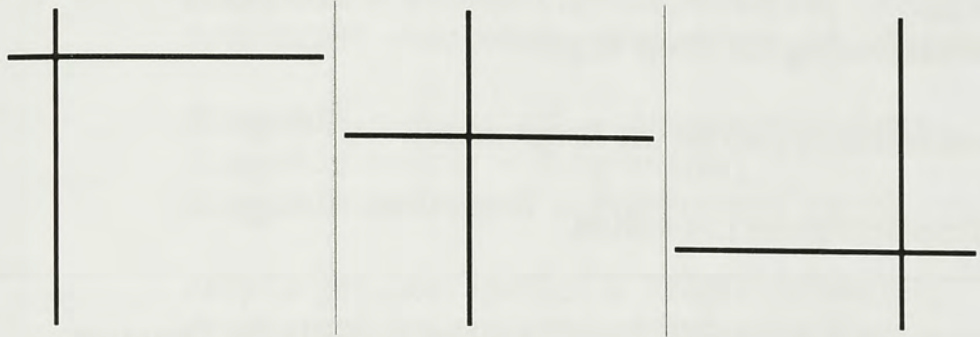


Figure 7-13. Three frames from a basic animation engine that sweeps two lines across a canvas

To see this working, edit `script.js` in the `canvas/js` directory. Obviously, this is just a simple example to show a bare-bones animation engine, but this structure could easily be developed further to create more complex animation sequences.

Drag-and-drop operations

Drag-and-drop functionality as it appears in the HTML5 specification is actually an old Microsoft addition,⁷ which was added to Internet Explorer originally more than a decade ago. Since then, the other browsers have caught up and implemented the same functionality—except for Opera, which does not currently support this feature natively. In the supporting browsers, drag-and-drop capability is available to all the elements through the global `draggable` attribute, which has the possible values of `true` or `false`. Most elements will be set to `false` by default, but there are a few exceptions. The `image` element (`img`) has its `draggable` attribute set to `true` by default, as does the `anchor` element (`a`), but only if it has its `href` attribute set. Also, although it doesn't have attributes, plain text that has been selected by the user can be dragged by default.

Dragging operations will always have a start and an end, which usually corresponds to something being picked up, moved, and dropped somewhere else. Quite a few events are involved, depending on the complexity of the desired behavior in between the start and end of a drag-and-drop sequence. Table 7-1 describes the relevant events and when they are triggered.

Table 7-1. Events involved in drag-and-drop operations

Event	Description
dragstart	The drag-and-drop operation has begun.
drag	The drag-and-drop operation is occurring.

⁷ See <http://dev.w3.org/html5/spec-author-view/dnd.html> for the HTML5 drag-and-drop documentation. Information about the Microsoft implementation it's inspired from can be found here: [http://msdn.microsoft.com/en-us/library/ms537658\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms537658(VS.85).aspx).

dragenter	The dragged item is entering the drop target.
dragover	The dragged item is over the drop target.
dragleave	The dragged item is leaving the drop target.
drop	The dragged item was dropped on the drop target.
dragend	The drag-and-drop operation has ended.

Let's create a file that logs each of these events to the console. Duplicate the `jstemplate` directory from earlier and rename it `dnd`. Begin by editing `index.html` to look like the following:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>HTML5 Apprentice</title>
    <link rel="stylesheet" type="text/css" href="css/styles.css" />
    <script type="text/javascript" src="js/script.js" defer></script>
  </head>
  <body>
    <section>
      <h1 draggable="true" id="draggable">Draggable</h1>
    </section>
    <section>
      <h1 draggable="false" id="droptarget">Drop Target</h1>
    </section>
  </body>
</html>
```

This HTML page creates two sections, one that has its `draggable` attribute set to `true` and one to `false` (this could be left off, but I included it to explicitly differentiate the two). Each has an ID that will be referenced in the JavaScript for the page.

Now edit the linked CSS file, `styles.css` (in the `dnd/css` directory). We will take the sections and create two black-bordered boxes that are arranged side by side:

```
section {
  width:200px;
  padding:10px;
  background-color:#cccccc;
  border:1px solid #000000;
  float:left;
  margin-right:4px;
  text-align:center;
}
```

Lastly, edit `script.js` (in the `dnd/js` directory) and write the following script:

```
// global variable to hold reference to the two sections on the page
var draggable;
```

```
var droptarget;

// initialize variable values and set event handling functions
function init() {
  draggable = document.getElementById( "draggable" );
  droptarget = document.getElementById( "droptarget" );

  draggable.ondragstart = dragStartHandler;
  draggable.ondrag = dragHandler;
  draggable.ondragend = dragEndHandler;

  droptarget.ondragenter = dragEnterHandler;
  droptarget.ondragover = dragOverHandler;
  droptarget.ondragleave = dragLeaveHandler;
  droptarget.ondrop = dropHandler;
}

// event handling functions for each of the drag and drop operations
function dragStartHandler(e) { console.log("dragstart"); }
function dragHandler(e) { console.log("drag"); }
function dragEndHandler(e) { console.log("dragend"); }
function dragEnterHandler(e) { console.log("dragenter"); }
function dragOverHandler(e) { console.log("dragover"); }
function dragLeaveHandler(e) { console.log("dragleave"); }
function dropHandler(e) { console.log("drop"); }
```

```
window.onload = init;
```

This script will add event handler functions for each of the drag-and-drop events and log a message to the console when the event occurred. If you open this in a web browser and drag the text “Draggable” to “Drop Target,” you should see some event messages appear in your JavaScript console.

Depending on the browser you used, you may not see very much. For instance, Google Chrome shows only “dragstart” and “dragend” and appears to ignore all the other events! What’s going on here? Well, Chrome expects something called a `dataTransfer` object to be set, which is essentially a data payload that gets (potentially) delivered to the drop target in a drag-and-drop operation. At its simplest, a drag-and-drop operation involves the following:

- Setting the `draggable` attribute to `true` (if it isn’t already)
- Setting an event listener function to handle the `dragstart` event
- Setting the payload of the `dataTransfer` object in the `dragstart` event handler function

OK, so we need to add a line of code to the `dragStartHandler()` function to make this script compatible with browsers such as Google Chrome. Amend the code for the `dragStartHandler()` function in `script.js` to look like this:

```
...
function dragStartHandler(e) {
  console.log("dragstart");
  e.dataTransfer.setData("text/plain" , "Payload Landed" );
}
...
```


As you can see, the `dataTransfer` object has a method named `setData()` that sets the type and value for the data that is sent along with the dragged item. Now (if you weren't seeing it before), you should see the following events logged to JavaScript console:

```
dragstart
drag
dragenter
dragover
drag
...
dragleave
dragend
```

In the middle, you will see “dragover” and “drag” repeated many times as you drag the item around.

You may notice one event is missing, the all-important drop! The reason for this is the `dragover` event has the odd behavior of resetting the drag-and-drop operation when you are over the drop target, which has the consequence that the drop event never fires. The `dragenter` and `dragover` events could be used to handle the drag-and-drop operation, but in most circumstances you will want to trigger the drop event. What you need to do then is cancel the default behavior of the `dragover` event. This is a matter of calling the method `preventDefault()` on the event object that is passed as a parameter into the `dragOverHandler()` function. Additionally, we will check whether the default behavior has already been prevented, and we'll also return false from the function, which is another way of telling the browser to ignore the default behavior of the event. Both methods can be added in order to ensure the broadest browser support for this instruction. Edit your code and add the following:

```
...
function dragOverHandler(e) {
    console.log("dragover");
    if (e.preventDefault) e.preventDefault();
    return false;
}
...
```

Now when you test the drag-and-drop operation, you should see an entry for “drop” in the JavaScript console. For the last edit, we'll assign the text of the “drop target” to the text in the payload in the `dataTransfer` object. We'll use the `innerHTML` property to set the content inside the drop target. Edit the script to make this change:

```
...
function dropHandler(e) {
    console.log("drop");
    droptarget.innerHTML = e.dataTransfer.getData("text/plain");
}
...
```

The `getData()` method is used to retrieve the contents of the payload set in the `dragStartHandler()` function, which then assigns that value to the contents of the drop target. Test the page again, and you should see that when you drop the “Draggable” text onto the “Drop Target” text it changes to “Payload Landed.”

OK, let's make this a little bit more user friendly. We're going to add a CSS class that creates a dashed black border that we'll make appear and disappear when over the drop target. We'll also add a default border to the `h1`s that blends into the background so the elements don't shift when the dashed border is added. Edit `styles.css`, adding the following two rules:

```
h1 {
    border:1px solid #cccccc;
}

.over {
    border:1px dashed #000000;
}
```

Now we'll add the `over` class when the drag is over the drop target by setting the `className` property on the drop target. Edit the `dragOverHandler()` function in `script.js` to add the class name:

```
...
function dragOverHandler(e) {
    console.log("dragover");
    droptarget.className = "over";
    if (e.preventDefault) e.preventDefault();
    return false;
}
...
```

We need to remove this class in the `dragEndHandler()` and `dragLeaveHandler()` functions, so we'll set the `className` property on those to null, which will remove the CSS class:

```
...
function dragEndHandler(e) {
    console.log("dragend");
    droptarget.className = null;
}
...
function dragLeaveHandler(e) {
    console.log("dragleave");
    droptarget.className = null;
}
...
```

Finally, we need to prevent the default behavior of the `dropHandler()` function. Some browsers will try to open content that has been dropped. For instance, a link that is dragged and dropped into a browser window may cause the browser to navigate to the link's URL. To prevent this, make the following edit to `script.js`:

```
...
function dropHandler(e) {
    console.log("drop");
    droptarget.innerHTML = e.dataTransfer.getData("text/plain");
    if (e.preventDefault) e.preventDefault();
    return false;
}
...
```

Now if you test this drag-and-drop operation, you should see the text payload added when the drag-and-drop operation is complete, and the border should appear and disappear during the operation (Figure 7-14).

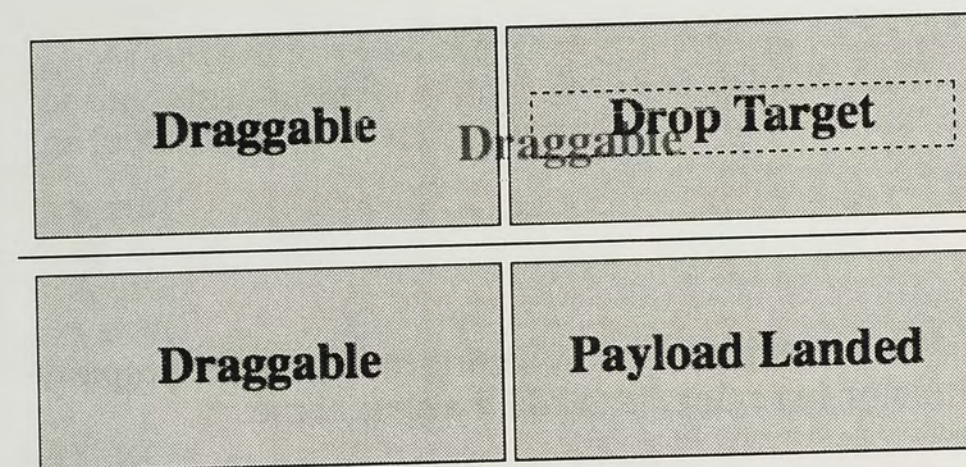


Figure 7-14. A drag-and-drop operation in progress (top) and completed (bottom)

In addition to the draggable attribute, there is the global dropzone attribute. This attribute can in theory be used to specify what kind of data a particular drop target expects to receive as well as what kind of drag-and-drop operation is going on (the specification lists move, link, and copy as its possible values), which in theory could be used to replace any logic introduced in the dragenter and dragleave events. However, at the time of writing, you'll have to wait for this feature to be implemented across the major web browsers.

ADDEVENTLISTENER

The syntax for handling events in this book has looked like the following, for example:

```
window.onload = init;
```

This is an older syntax that I've used because of its wide support, but you should be aware of a newer, more flexible syntax for handling events that you are likely to run into. The `addEventListener()` method can be added to associate an event handler function with an event. The preceding line of code could be rewritten as follows:

```
window.addEventListener( "load" , init , false );
```

This associates the load event with the function `init()`, so in this syntax the "on" prefix is left off the event. The advantages of this syntax are as follows:

- The false in the previous code snippet controls exactly when to respond to the event. In short, events can be triggered during two phases: the capture phase and bubbling phase. In the capture phase, a particular element would trigger an event after all its parent elements in the DOM had triggered the event. The bubbling phase is the opposite; a particular element triggers the event before its parent containing elements. A value of true means the event listener responds during the capture phase, and a value of false means it responds during the bubbling phase. The value is usually set to false.
- By adding additional calls to `addEventListener()`, you can associate more than one function with a single event.

Sorting a list using a drag-and-drop operation

Let's look at another example of a drag-and-drop operation, in the form of a user sortable list. For this example, you'll create an ordered list where the user can—through a drag-and-drop operation—rearrange the order of the list items.

To begin, duplicate the `jstemplate` directory from earlier and rename it `dnd_list`. Edit the `index.html` file to look like this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>HTML5 Apprentice</title>
    <link rel="stylesheet" type="text/css" href="css/styles.css" />
    <script type="text/javascript" src="js/script.js" defer></script>
  </head>
  <body>
    <ol>
      <li>One</li>
      <li>Two</li>
      <li>Three</li>
      <li>Four</li>
      <li>Five</li>
    </ol>
  </body>
</html>
```

Edit the JavaScript file named `script.js` in the `dnd_list/js` directory. This script will use four functions, `init()`, `dragStartHandler()`, `dragOverHandler()`, and `dragEndHandler()`. First create the global variables and edit the `init()` function:

```
// define global variables
var dragging;           // the list item being dragged
var dir;                // the direction (up or down) the drag is going in

function init() {
  var li = document.getElementsByTagName("li");
  // loop through list and make items draggable and set the event handler functions
  for ( var i = 0; i < li.length; i++ )
  {
    li[i].draggable = true;
    li[i].ondragover = dragOverHandler;

    li[i].ondragstart = dragStartHandler;
    li[i].ondragend = dragEndHandler;
  }
}
```

This example differs from the previous one in that the `draggable` attribute value is being added from the script. The script retrieves a reference to all list item elements on the page and adds the ability for them to be swapped around.

Continue with the script:

...


```
function dragStartHandler(e) {
    dragging = e.target;
    e.dataTransfer.setData('text/plain' , null);
    dragging.style.opacity = "0.5";
}
```

The `dataTransfer` object is set here but to a value of `null` because we're not using a payload. It just needs to be set to allow the `dragover` event to fire in Chrome. We're setting the CSS `opacity` property here too, although it would be advisable to add and remove a CSS class like in the earlier example so that you could easily add more styles in the future. For brevity we'll leave it as is in this example.

The `dragOverHandler()` function does the real work in this example:

```
function dragOverHandler(e) {
    // make sure the item being dragged isn't the same as the one we're dragging over
    // and make sure it hasn't been removed from the page
    if (dragging != e.target && dragging.parentNode != null)
    {
        // determine whether the drag is going up or down
        if ( e.target.previousElementSibling == dragging ) dir = "down";
        else dir = "up";

        // remove the item being dragged from the page
        dragging.parentNode.removeChild(dragging);

        // add item being dragged above or below the item being dragged over
        if (dir == "down"){
            dragging = e.target.parentNode.appendChild(dragging , e.target);
        }
        else if (dir == "up")
            dragging = e.target.parentNode.insertBefore(dragging , e.target);
    }
    // prevent the default behavior
    if (e.preventDefault) e.preventDefault();
    return false;
}
```

This function contains the logic for removing the dragged item and adding it back in at a different spot.

Add the final function that resets the styles and, lastly, ensure the script is set to run when the page fully loads:

```
function dragEndHandler(e) {
    dragging.style.opacity = "1.0";
}
```

```
window.onload = init;
```

That's it! You should have an ordered list that you can drag and drop the items in to change the order (Figure 7-15).

1. One
2. Three
3. Four
4. Two
5. Five

Figure 7-15. A drag-and-drop ordered list allowing for user reordering of the list items

Summary

My hope is that this chapter has shown you how you can peer into the JavaScript APIs that are sprinkled throughout the HTML5 specification. With scripting, the possibilities are as big as your imagination. The page can be changed in so many ways, dynamically and interactively. For example, an initially uninspiring thing like a blank canvas can be built to include all sorts of interesting effects for generating real-time changes to images. Or drag-and-drop functionality can be used for building all sorts of unique interfaces that might be found on such applications as web forms, games, or administrative interfaces. Using JavaScript, HTML, and CSS together, you could create beautiful skins (controls) for a custom HTML5 video player, which would look the same across all major web browser platforms. You could take the same approach with the audio element as well. The best way to learn how the JavaScript APIs work is to use them; and remember, if something doesn't work, take a step back and use the output in `console.log()` to help familiarize yourself with what is going on and what is available for you to use.

CHAPTER 8

The Road Ahead

There are many fringes to HTML5, areas that are still being molded and sculpted toward a standard perfection, but to a large degree the road ahead has less to do with HTML5 and more to do with where HTML5 will be used. It is a connected world, and more people are connected via cellular phones than desktop computers. In the years ahead, Internet-enabled smartphones will likely spread to the masses, like low-end “dumb” mobile phones did years ago,¹ and the Web will have visitors who are on the move, surfing web pages on crowded buses, on street corners, in cafés, in restaurants, and, dare I say it, in movie theaters. Mobile devices offer special challenges for you, the web developer/designer. The screen resolution is more complicated, and the screen real estate is more confined, but with a helping hand from CSS3, HTML5 is well supported on contemporary smartphones. What you learned from prior chapters applies here, albeit things may be a little slower and there is a smaller screen area to work with, but those are the design challenges to work around, and they are counterbalanced by the unique possibilities of the mobile space. This chapter will discuss those challenges and possibilities and finish with a discussion of some last corners of the HTML5 specification.

Challenges of the mobile Web

Over the past few years, a host of new Internet-enabled handheld devices have become widely available. While Internet-enabled mobile devices have been around since the 1990s, they arguably first became widely viewed as a viable platform for viewing web pages (as opposed to just using them for e-mail, for instance) with the introduction of the iPhone and its large colorful touchscreen in 2007. What followed was an explosion of large touchscreen devices running iOS, Google’s Android OS, Microsoft’s Windows Phone 7 platform, and more. Designing for the Web suddenly became a multifaceted process, many times more so than before. A web page might be viewed on a large wide-screen display in a desktop computer environment with loads of processing power, but it may be viewed on a small handheld mobile phone, with processor constraints and a display of just a few inches across. And then there are tablet devices that fall in between the two, such as Apple’s iPad or the Samsung Galaxy Tab. However, designing for more than desktop displays is about more than taking differences in screen size and processing power into account. Mobile devices won’t have a mouse, and clicking will likely be done with a finger (or stylus if the device is older), so small buttons are best avoided. Then there is the issue of connectivity. Mobile devices may drop their connection as the user is moving about.

¹ To appreciate the explosive growth in the worldwide use of cellular devices, take a look at the data collected by the World Bank on mobile cellular subscriptions:
<http://data.worldbank.org/indicator/IT.CEL.SETS.P2/countries/1W?display=graph>.

Responsive design

Let's consider screen size to begin with, because it is the most obvious difference between desktop computers and handheld devices. There are two broad approaches to mobile design. The first is to essentially develop two different sites, one for a desktop environment and another one for a mobile device. The user is directed to one or the other depending on what browser they view the site in. This may be a viable approach for complex sites that may need lots of effort to accommodate a mobile viewing experience and are better off presenting a simpler version of the site for mobile users. A problem with this approach is the potentially high maintenance involved. If a different site is created for desktops, tablet devices, and mobile phones, what happens if another device with a different form factor becomes popular? For example, Opera has a web browser for the Nintendo Wii game console, and Internet-enabled wristwatches running Android OS have even been developed! The other option is to change the style sheet in use based on the screen size of the viewing device. **Media queries** (discussed a little later) can be used to change the CSS depending on the physical qualities of the viewing device, such as changes based on the display size and resolution. Combine this approach with a fluid layout, and you get what is termed **responsive design**,² which aims to accommodate a broad range of screen sizes.

A fixed 960-pixel-wide grid has long been a web design standard that uses CSS to create rows and columns. Developed from this has been responsive grid design, which increases or decreases the amount of columns on the page based on the width and expands the columns visible to the width of the browser viewable area. Development kits such as Skeleton (<http://getskeleton.com>) have emerged to help develop flexible layouts of this sort.

■ **Note** Adaptive images (<http://adaptive-images.com>) is a concept related to responsive design that is for handling the scaling of images on a page so they scale down for mobile devices and scale up for desktop displays.

The viewport

On a desktop environment, the **viewport** is the area of a web page visible in a web browser. For instance, if you opened up a web page on your desktop and resized the page down to the physical size of a mobile phone display, the area you see is the viewport. You will likely see scrollbars appear so you can scroll to the rest of the content. You'll also notice the text may wrap to fit in the visible area, if it is not otherwise sized with CSS.

The behavior of the viewport on a mobile device is slightly different. Unlike a desktop environment, in a mobile environment, the web browser covers a fixed area. The viewport defines the area in which text would be wrapped and so forth, which may extend outside the viewable area. The default viewport on iPhone, for instance, is 980 pixels wide (to comfortably accommodate the 960-pixel web page width commonly used in web design), but the page may be scaled down so it fully fits in the 320-pixel viewable area (Figure 8-1).

² See the seminal article by Ethan Marcotte on responsive web design at <http://alistapart.com/articles/responsive-web-design/>.

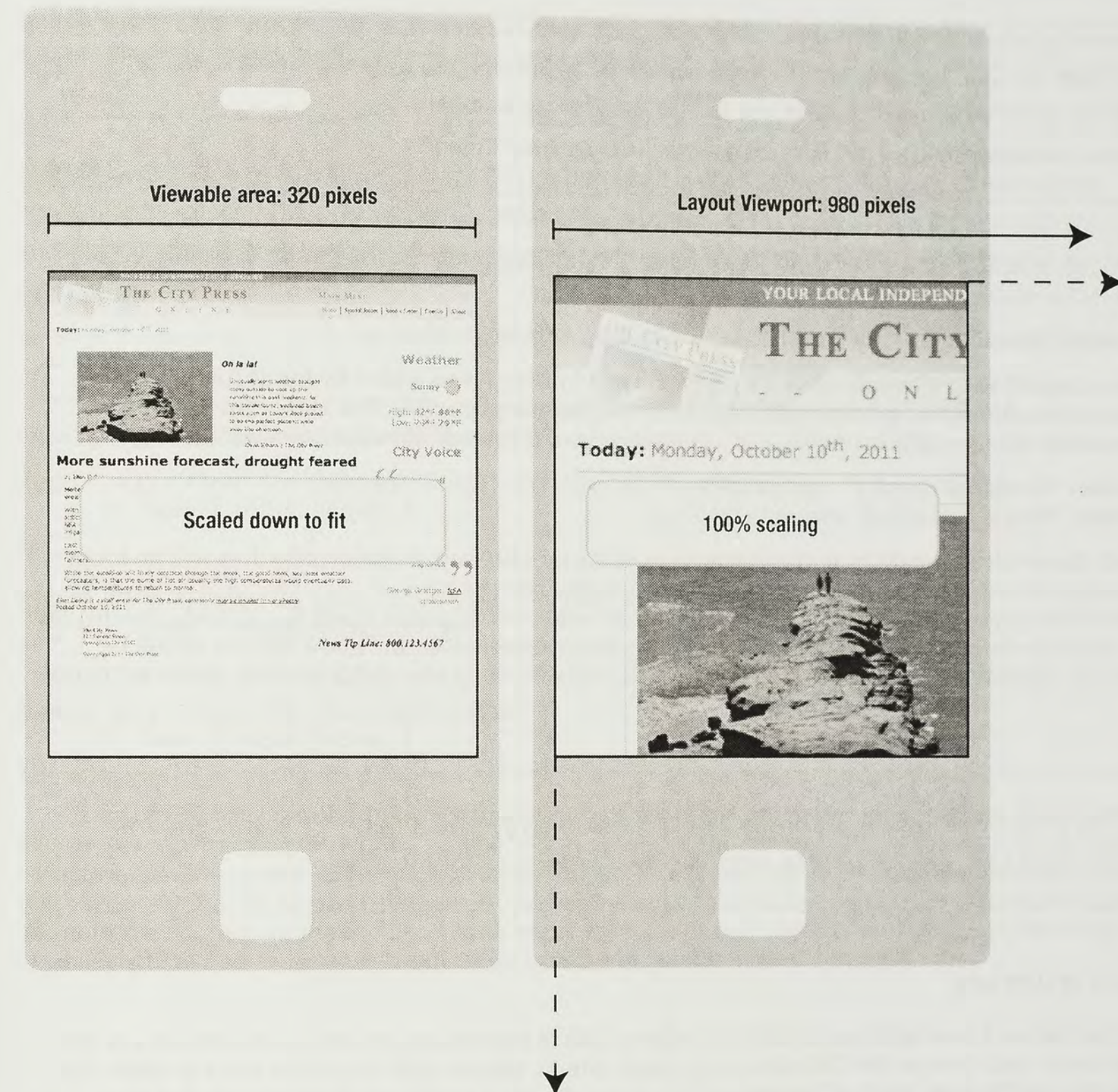


Figure 8-1. The viewable area on a typical smartphone is 320 pixels, while the viewport the page is laid out in may be much larger, requiring the user to pan to see all content on the page when viewing at actual size.

Ideally, the web page width would correspond to the viewable area, and this is what can be done to optimize a web page for mobile viewing. To begin with, there is a meta element value that tells a mobile device to set the width of the layout viewport to the width of the viewable area. To do this, the following would be added in the head section of the web page:

```
<meta name="viewport" content="width=device-width" />
```

■ **Note** There is a CSS rule, `@viewport`, that is under development as an alternative for the meta element `viewport` rule. For implementation details, see this page on Opera's website:
www.opera.com/docs/specs/presto28/css/viewportdeviceadaptation/.

Next, an additional option can be added to prevent the page from zooming unnecessarily. Add `initial-scale=1.0` to the meta element:

```
<meta name="viewport" content="width=device-width, initial-scale=1.0" />
```

The viewport meta element value is a fairly recent addition; it was added by Apple for use on the iPhone but has since been incorporated into other contemporary mobile OSs. However, to accommodate older mobile browsers, it is a good idea to add two additional meta elements:

```
<meta name="HandheldFriendly" content="true" />
<meta name="MobileOptimized" content="320" />
```

Both these values herald from the earlier days of the mobile Web, so eventually they will no longer be necessary to include, but for now it is a good idea to include them.

Now if the page layout is optimized for a 320-pixel-wide viewing area, it will fit right into the display area of most modern mobile phones, but to actually achieve this width and allow the site to still be comfortably viewable in a desktop environment, you will very likely use media queries, which we'll look at next.

■ **Note** The HTML5 Boilerplate project mentioned in Chapter 1 has a version tailored for mobile devices at <http://html5boilerplate.com/mobile>, which includes these meta element values.

Media queries

Media queries are a new addition in CSS3 that allows CSS to inspect the physical characteristics of the viewing device and change the CSS rules accordingly. Media queries grew out of the need to tailor CSS for different media types, such as for print or for screen. A typical media query might look like this:

```
@media only screen and (min-width:480px) and (max-width:767px) {
    body { width:560px; }
}
```

This appears in an external CSS file linked into the page. It specifies both the type of media the enclosed CSS rules will apply to (in this case it's for displaying on a screen) and the minimum and maximum width of the viewing screen on which the enclosed rules would apply. It's not necessary to

specify both the minimum and maximum widths. A lot of other values can be selected against as well, such as an exact width, height, orientation, color, resolution, and so on.³

■ **Note** Since media types (screen, print, etc.) are supported in older browsers, the `@media` rule will not necessarily be ignored. The only keyword is added at the beginning of the query to hide the enclosed style rules from older browsers, because they will not recognize it, which will cause them to skip past the rest of the media query declaration.

Using media queries in a tiered configuration is useful for accommodating ranges of screen sizes, each fitting the content into a smaller and smaller screen area:

```
@media only screen and (min-width:768) and (max-width:959px) {
    body { width:768px; }
}

@media only screen and (min-width:480) and (max-width:767px) {
    body { width:480px; }
}

@media only screen and (max-width:479px) {
    body { width:340px; }
}
```

These three rules constrain the width of the page through three screen size changes. This example just shows constraining the width, but you will want to style the content all-around for mobile viewing, which would likely mean larger text and larger link areas. For example, the *City Press* news tips form used in Chapter 4 might be altered using media queries to appear in a mobile browser, as shown in Figure 8-2. To tailor the content for the mobile environment, extraneous information has been positioned out of view using CSS, and styles have been added to make the menu more prominent.

³ See www.w3.org/TR/css3-mediaqueries/ for the full list and details.

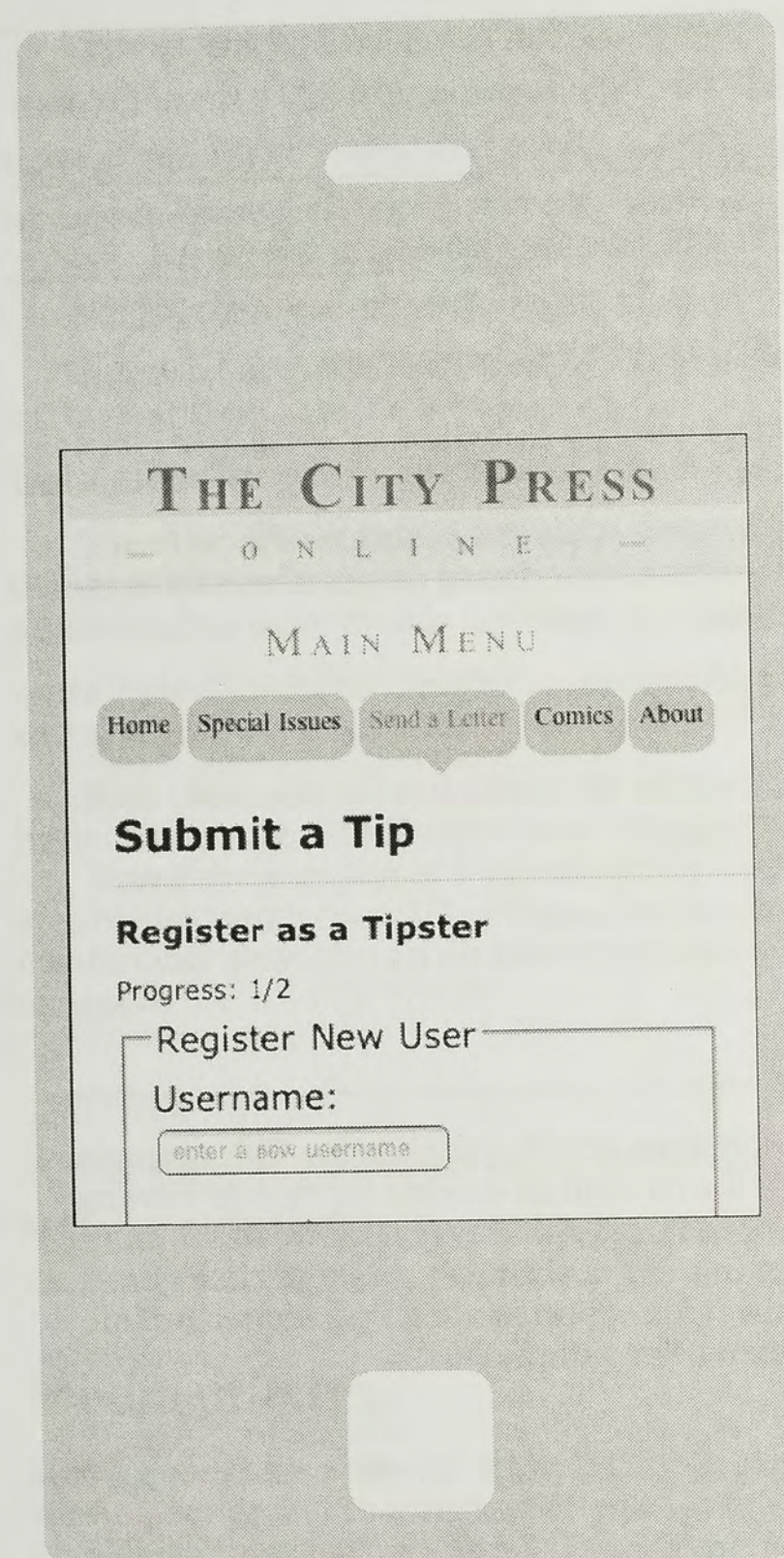


Figure 8-2. The City Press mobile-optimized news tip submission form page

TESTING MOBILE WEB PAGES

Testing a web page on an actual mobile device is the best way to know that a mobile page is, in fact, optimized for a mobile device, but it's also possible to quite effectively test web pages on emulated devices running on a desktop computer. Each major platform has its own simulated device that runs on a desktop computer. This is great for checking the layout and basic functionality but is not so great for testing performance, because the simulated device is using the processor of the computer, which is many

times more powerful than the processor in the mobile device. At any rate, the following are some resources you may find useful for testing mobile web pages on a desktop computer:

- **Android:** The Android emulator can be downloaded as part of the Android software development kit (SDK) at <http://developer.android.com>, and an assortment of hardware specifics can be emulated and simulated. One caveat of note with Android is that when visiting a page on a local web server, the usual <http://localhost/> address will look for a web server running on the emulator itself. Instead, the special URL <http://10.0.2.2/> needs to be used to access a web server running on the host computer.
- **iOS:** Apple includes a simulator with its Xcode software package, available at <http://developer.apple.com/xcode/>. The simulator can simulate an iPad and an iPhone with and without a Retina display.
- **Windows Phone 7:** The Windows Phone SDK includes an emulator for testing content on the Windows Phone 7 platform. This is downloadable from <http://create.msdn.com>.
- **Opera:** Opera doesn't develop hardware but has a mobile version of its desktop web browser, which has a simulator that may be downloaded from <http://opera.com/developer/tools/mobile>.

In addition to this list, Adobe Device Central is an option for creating generic virtual mobile devices of various configurations and screen sizes. A web page URL can be opened in a virtual device under File ► *Open URL....*

It's also possible to test a responsive design in a standard web browser. If the Develop menu is turned on in the advanced preferences of Safari, you will find that there is an option under Develop ► User Agent to have the browser masquerade as a mobile browser (or even a browser from another manufacturer!). Open a web page and resize the window, and it'll act like it is being viewed on Safari's mobile version!

Offline application cache

The offline application cache is a way to ensure that critical files are stored on the client so that if the connection is lost and the page is reloaded, the cached files will still be loaded. If all necessary files are cached, the website can be navigated normally even when the user is offline. This may be particularly useful in a mobile web context, where the Internet connection might be intermittent.

The application cache uses the manifest attribute found on the root html element to associate the web page with a cache manifest file that lists the files to cache (or in some cases, not cache!). The cache manifest file is just a text file with the .manifest file extension. It must be served from the web server using the text/cache-manifest MIME type. It uses a relative or absolute URL but typically looks something like this:

```
<!DOCTYPE html>
<html manifest="/sitecache.manifest">
  <head>
    ...
```

The sitecache.manifest file (depending on your configuration; you may name it differently) must begin with the key phrase CACHE MANIFEST and then is followed optionally by three sections. These are the following:

- **CACHE:** The files to cache.
- **NETWORK:** The files that always should be retrieved from online.
- **FALLBACK:** Fallback content for files that were not found in the cache. The file to find is given first, followed by the fallback file's location.

The cache manifest can also contain comments, which are ignored. Comments begins with the hash sign (#). A cache manifest file might look something like this:

CACHE MANIFEST

```
# list of files to cache
CACHE:
index.html
css/styles.css
js/script.js
```

```
# files that require a network connection
NETWORK:
userlogin.php
```

```
# files to use in place of other files that were not cached
FALLBACK:
images/logo.png images/offline-logo.png
contact.php contact.html
```

This would need to appear on any page that was using the cache. The files listed under the CACHE section will be stored on the client machine and retrieved as necessary to navigate the site. Any files appearing under the NETWORK section will always be fetched from the server. The FALLBACK section maps pairs of files together so that if the first one is not found, the second one will take its place on a page. Using fallbacks, you can provide visual clues to the user that they are surfing a cached version of the page. For example, you could provide alternate images that are branded “offline,” which would show up only when the user was viewing the cached images.

Other HTML5 technologies

There are a number of technologies in the HTML5 family that either are not mature enough or are too broad to address within the scope of this book. However, that is no reason you shouldn't be aware of their existence! What follows is a high-level overview of these technologies.

Microdata

Microdata is HTML5 for machines. It's for search engines and other external applications that may want to derive meaning from your content but need more information to understand it. To do this, existing HTML elements can be annotated with attributes that define the type of data on the page to a much finer degree than what is possible with existing semantic HTML elements.

At its most basic level, Microdata uses three attributes: `itemscope`, `itemtype`, and `itemprop`. Consider, for instance, a paragraph about this book:

```
<p>
HTML5 Mastery: Semantics, Standards, and Styling by Anselm Bradford and Paul Haine.
</p>
```

To mark this up using Microdata, first the `itemscope` property is added to indicate that this paragraph contains a collection of data that is related, a book and authors in this case:

```
<p itemscope>
HTML5 Mastery: Semantics, Standards, and Styling by Anselm Bradford and Paul Haine.
</p>
```

This annotation means that any elements inside the paragraph are related to each other. The `itemtype` attribute is then used to give a URL of a Microdata vocabulary, which is a defined set of property values for different types of common data, such as people, places, events, and things. The website <http://schema.org> (which was launched by Microsoft, Google, and Yahoo!) contains a number of vocabularies for an assortment of data. We'll use their documentation on a Book to annotate this paragraph. The `itemtype` is added with the URL to the vocabulary definition:

```
<p itemscope itemtype="http://schema.org/Book">
HTML5 Mastery: Semantics, Standards, and Styling by Anselm Bradford and Paul Haine.
</p>
```

Visiting the vocabulary address, you will see there are a large number of properties that could be used, many more so than what we need in this simple example. We'll use the `itemprop` attribute to annotate the content with properties from the documentation. To separate the title from the authors, we'll also need to add span elements for the annotations to attach to:

```
<p itemscope itemtype="http://schema.org/Book">
<span itemprop="name">HTML5 Mastery: Semantics, Standards, and Styling</span> by <span
itemprop="author">Anselm Bradford</span> and <span itemprop="author">Paul Haine</span>.
</p>
```

First the name of the book is annotated, followed by the authors. The authors are a bit of a special case, because they are another object in the Microdata universe, that of a Person, which means they could have their own `itemscope` property and have additional information marked up about them, such as birth date, address, telephone number, nationality, and so on.

Search engines or other applications could parse this paragraph and be able to pick out where the book title began and ended and who the authors are. This is just a taste, but you should now have an idea of what Microdata is about. The concept is rather simple: keys and values are used to provide detailed metadata for content on a page.

There's some contention over Microdata, because it overlaps with two other formats that existed before it, Microformats and RDFa⁴ (Resource Description Framework in attributes). Also, since <http://schema.org> was launched by several large corporations with vested interests in search engine technology, the neutrality of the syntaxes advocated by this website have been questioned. It is, however, a good starting point to familiarize yourself with the concepts of annotating your page content.

⁴ Microdata actually evolved from RDFa.

Undo manager API

The undo manager API is a new interface for allowing a web page access to the undo/redo transaction manager inside the web browser. What does this mean? Well, think about when you type some text in the search bar in a web browser; you will likely be able to select Edit ► Undo to undo the text that you entered. This is the browser’s native undo manager allowing you to undo an operation you have done. However, if you have an interactive web application, such as a rich text editor or drawing application built on canvas, which is controlled by JavaScript, the browser’s native undo manager would not pick up the modifications performed by the script. This is where the undo manager API would come in, because it would provide a method to add actions performed via JavaScript to the browser’s undo/redo manager. Just as the history API operates by adding URLs to the history “stack,” the undo manager allows a particular set of changes to the page (together called a transaction) to be added to the browser’s native undo stack. Since this feature is still under development, don’t expect it to work in browsers today, but keep it in mind as a capability that will likely arrive in browser implementations in the not-too-distant future.

Upcoming CSS technologies

As you know, this one isn’t an HTML5 technology, but it goes hand in hand with HTML. As there are many developments in HTML5, there are also many in CSS3. An interesting area has to do with page layout, and the specifications in this area have yet to fully coalesce into a dominant method(s). Eventually it’s expected there will be three main specifications in this area: multicolumn layouts (discussed in Chapter 6); Flexible Box Layouts,⁵ or FlexBox, which provides better alignment and positioning of elements within their parent container; and grid/template/region layouts. Currently, grid/template/region layouts are broken into four distinct specifications, but these will likely merge in the future. **Grid positioning and grid layouts** are for handling rows and columns and specifying how elements are positioned in this type of layout (a multicolumn layout can already be thought of as a rowless grid layout). The **template layout** uses a grid as well but defines a template such as “abc” or “bca,” where each of the letters corresponds to an element, which can then be rearranged by changing the template lettering order. Lastly, the **region layout** proposes a means by which content would “flow” from one element to another when it overflows its bounds. Positioning in CSS has always been a headache, particularly for novice designers, so these new proposals to the specification will likely empower CSS coders in the same way that HTML5 has provided an explosion of possibilities for HTML coders (and very likely these two people are one and the same, so the creative possibilities will likely be dramatically expanded!).

Lastly, it’s worth noting some of the new selectors that are defined in the Selectors Level 4 specification.⁶ See Table 8-1 for a list of the upcoming new selectors and a description of their use. Don’t expect these to work in the near future, but down the road these will likely be supported by your preferred web browser.

⁵ See www.w3.org/TR/css3-flexbox/.

⁶ See <http://dev.w3.org/csswg/selectors4/>.

Table 8-1. The new selectors described in the Selectors Level 4 specification

Selector	Description
:matches(S)	Any element that matches one or more compound selectors in a list of selectors (S).
:any-link	Any hyperlink anchors.
:local-link	Any hyperlink anchors that link within a site. Allows internal links to be differentiated from external links that go to other website domains.
:scope	Equivalent to :root unless a contextual reference element set is given. What does that mean? It’s a group of elements returned by the <code>querySelector()</code> method in JavaScript, for instance.
:current	In a time-dimensional rendering of the HTML document, such as when text-to-speech is being used to read the document, this selects the element currently being processed.
:past	A previous sibling of the an element selected with :current.
:future	A sibling occurring after an element selected with :current.
:dir(L)	Directional selector selects either left-to-right running text with :dir(ltr) or right-to-left running text with :dir rtl).
:nth-match(S)	An element that is the Nth sibling before an element that matches a compound selector in a list of selectors (S).
:nth-last-match(S)	An element that is the Nth sibling after an element that matches a compound selector in a list of selectors (S).
:nth-column(N)	An element that is the Nth sibling before a cell element belonging to a column in a grid of rows and columns.
:nth-last-column(N)	An element that is the Nth sibling after a cell element belonging to a column in a grid of rows and columns.
:column(S)	A cell element belonging to a column in a grid of rows and columns that is specified by a selector (S).

Summary

Predicting the future is easy if you assume that the direction things were headed in was wrong and set out to define your own new path. And it seems this has been the way HTML5 has turned out. Web developers disillusioned by the confines of XHTML turned to HTML5 with the hope that it would provide the platform they needed for a new generation of web applications. Now that it has paved its own path forward, the momentum to reinvent the Web has brought a whole new palette of possibilities. There is the flexibility of syntax, the new elements, and the multimedia and richness of JavaScript and CSS features. We've covered a lot, and there is more to come, but for now we've come to the end of this journey. Let this be the beginning of your exploration. There are many roads you can take from here, such as delving deeper into mobile development, CSS3, or the extensive HTML5 APIs. The details of the specifications will no doubt evolve, because that is the nature of the Web, but the fundamentals of what you learned here and the concepts behind the components of HTML5 will continue under the same foundation of knowledge, even if there are changes in the syntax in the future. It is time to play with these technologies and see what is possible in the world of open standards. Go on and craft pages that are semantically sound and rich in content. Go on and make the Web a better place.

APPENDIX

Associated Technologies

In addition to CSS3, a number of technologies have been associated with HTML5 but actually are defined in their own specifications. Some were once part of HTML5 but have been spun off into their own documents as they have grown, while others are directly used with HTML5 but were never part of the HTML5 specification. This appendix provides a high-level overview of these different technologies that are commonly associated with HTML5. Additionally, you will also find a list of useful website resources for working with HTML5 at the end of the chapter.

Geolocation

The Geolocation API¹ defines how a web browser can determine a user's geographic location, using their IP or WiFi address or, if they are on a mobile device, the Global Positioning System (GPS) on the device. The location is given as a latitude and longitude coordinate, which may be more or less accurate, depending on the method used in retrieving the location. Determining a user's location with their IP address, for instance, will be far less accurate than using the satellite-based GPS to determine their location.

The implications of retrieving a user's location are quite fascinating, because it enables location-based services to be provided through a web browser. This is perhaps most useful for mobile devices, because information and advertisements can be delivered to a web browser on a device that is on the move.

The JavaScript methods for retrieving a user's location are held in the Geolocation object, which is inside `window.navigator.geolocation`. The location can be retrieved one time or can be updated continuously. The API defines three methods:

- `getCurrentPosition()`: Retrieves the current position one time
- `watchPosition()`: Retrieves and updates the current position as it changes
- `clearWatch()`: Stops updating a watched position

When attempting to retrieve the user's location, the web browser will typically provide a prompt as to whether using geolocation is allowed.

¹ See www.w3.org/TR/geolocation-API/.

Retrieving the current position

The `getCurrentPosition()` method is given a parameter for the function to call when the position has been successfully obtained (which could take several minutes if GPS is used). Here's an example:

```
function init() {
    // get the current position and call the "locatedSuccess" function when successful
    window.navigator.geolocation.getCurrentPosition(locatedSuccess);
}
function locatedSuccess(geo) {
    // log the returned Geoposition object
    console.log(geo);
}
window.onload = init;
```

■ **Note** Depending on the web browser used, this code will likely work only on a live web server (one running locally is fine). If it is not working, first check that the URL address of the page includes `http://` at the beginning.

This script will attempt to obtain the current location and then call the `locatedSuccess()` function when it has done so. The location query is done asynchronously so that other processes can continue to function on the page. The function is handed a parameter that contains a `Geoposition` object that contains information about the location. The `Geoposition` object contains a `timestamp` property and `coords` property, which contains yet another object, a `Coordinates` object. The `Coordinate` object contains the following properties, which may contain null values depending on the hardware capabilities of your viewing device (for instance, if your device does not have GPS capabilities, these values will be limited):

- `latitude`: The north-south position on the earth
- `longitude`: The west-east position on the earth
- `altitude`: The height of the position, gathered if the viewing device has the capability to measure altitude
- `accuracy` and `altitudeAccuracy`: The accuracy of the position as measured in meters
- `heading`: The direction of travel as measured in degrees around a circle
- `speed`: The speed of travel in a certain heading in meters per second

■ **Note** In addition to the `timestamp` and `coords` properties, Firefox includes an `address` property for retrieving address information such as city, country, and even street information!

Update the preceding `locatedSuccess()` function to print the location data on-screen:

```
function locatedSuccess(geo) {
    var lat = geo.coords.latitude;
    var long = geo.coords.longitude;

    document.body.innerHTML = "<ul><li>lat:"+lat+"</li><li>long:"+long+"</li></ul>";
}
```

An additional function name can be given to the `getCurrentPosition()` method to specify a function to run when the request for the user's location has failed. Edit the `init()` code and add a `locatedFailed()` function:

```
function init() {
    // get the current position
    //and call the "locatedSuccess" or "locatedFailed" if
    successful or not
    window.navigator.geolocation.getCurrentPosition
    (locatedSuccess, locatedFailed);
}
function locatedFailed(e) {
    // log the error code and message
    console.log(e.code , e.message);
}
```

The `locatedFailed()` function will run when the location was unable to be obtained. The parameter handed to it is a `PositionError` object that contains an error code and a message. The following are possible errors:

- *Error code 1, permission denied*: The user didn't authorize using geolocation.
- *Error code 2, position unavailable*: The position can't be determined.
- *Error code 3, position retrieval timed out*: Retrieving the position took too long.

If you want to test this function, the easiest way is to deny the geolocation request from the browser. Depending on the browser and whether you accepted the previous request for geolocation information, the browser will remember your choice and won't ask you again. Google Chrome, for instance, will require you to click the "target" icon in right of the address bar where you will have the option to clear the geolocation settings (the page will need to be reloaded for the settings to take effect). Figure A-1 shows what this dialog looks like. For Firefox, the geolocation permissions are found under Tools ► Page Info, which will open a dialog box with information about the page currently being viewed. Selecting the Permissions tab will allow you to set the preferences for sharing location information with a particular page. Safari has location settings set in its Privacy tab in the browser's Preferences pane. If you are unsure of where to clear geolocation information in your preferred browser, check in the application's preferences or the right side of the address bar, because these are the usual locations for the geolocation permission settings.

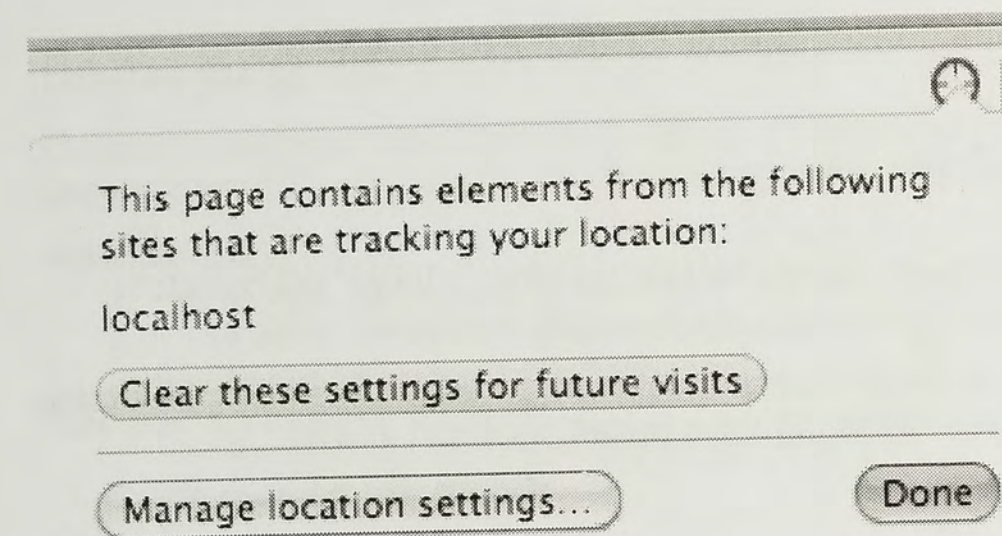


Figure A-1. Dialog for clearing geolocation settings for a page being viewed in Google Chrome

Lastly, the `getCurrentPosition()` method can be handed a custom object that can be used to set various options used in retrieving the location. This object can be set up with the following properties and values:

- `enableHighAccuracy`: Set to true or false. If enabled, the highest accuracy method for determining the location is used, such as GPS. Be aware that this will increase battery usage and the length of time for retrieving the location.
- `timeout`: How long to wait (in milliseconds) when retrieving the location before throwing a position unavailable error.
- `maximumAge`: How long (in milliseconds) a particular position should be considered the current position.

These options may be added to the `getCurrentPosition()` method using shorthand object creation notation, like so:

```
var options = {
  enableHighAccuracy: true,
  timeout: 120000,
  maximumAge: 1000
};
window.navigator.geolocation.getCurrentPosition(locatedSuccess, locatedFailed, options);
```

This will enable high-accuracy positioning (which is dependent on the hardware available), set the timeout to two minutes, and set the maximum age of the location to one second.

Watching the current position

Getting the location once is fine for a stationary device, such as a desktop computer, but for a mobile device, the location would have to be continually retrieved to be accurate. The `watchPosition()` method is used to continually poll the location (this is where the maximum age option is useful) to update the location information. It takes the same arguments as the `getCurrentPosition()` method, but it should be set to a variable that can later be referenced and handed to the `clearWatch()` method if updating the location continuously is stopped. Here's an example:

```
var geoWatchID = window.navigator.geolocation.watchPosition(locatedSuccess, locatedFailed, options);
```

Later in the code, `geoWatchID` can be passed as an argument to `clearWatch()` to stop the position from updating:

```
clearWatch(geoWatchID);
```

SVG and MathML

SVG and MathML have two totally different purposes, but they have one thing in common: they are both XML-based languages that can be embedded in HTML5. Scalable Vector Graphics (SVG) is for describing vector shapes, while Mathematical Markup Language (MathML) is for describing mathematical notation.

SVG is one half, along with canvas, of the Web's standard imaging options. While canvas deals well with bitmaps, SVG deals well with vector shapes. It also has built-in animation capabilities, which would need to be built from scratch in canvas.

The syntax of both is beyond what can be covered here, but being XML-based, they both look very much like HTML, except with a different set of elements. For example, the following code shows an HTML page that includes both MathML and SVG to describe and diagram the trigonometric functions shown in Chapter 7.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>SVG and MathML Demo</title>
  </head>
  <body>
    <h1>SVG and MathML embedded in an HTML5 page</h1>
    <p>
      <math>
        <mi>x</mi>
        <mo>=</mo>
        <mrow>
          <msub><mi>x</mi><mn>1</mn></msub>
          <mo>&plus;</mo>
          <mi>cos</mi>
          <mfenced><mi>&#x3B8;</mi></mfenced>
          <mo>&InvisibleTimes;</mo>
          <mi>c</mi>
        </mrow>
      </math>
    </p><p>
      <math>
        <mi>y</mi>
        <mo>=</mo>
        <mrow>
          <msub><mi>y</mi><mn>1</mn></msub>
          <mo>&plus;</mo>
          <mi>sin</mi>
          <mfenced><mi>&#x3B8;</mi></mfenced>
          <mo>&InvisibleTimes;</mo>
          <mi>c</mi>
        </mrow>
      </math>
    </p>
  </body>
</html>
```



```

</math>
</p><p>
<svg>
  <circle r="100" cx="101" cy="101" fill="white" stroke="black"/>
  <polygon points="101,101 171.710678,30.2893219 171.710678,101"
style="fill:white;stroke:black;" />
  <rect width="10" height="10" x="161.710678" y="91"
style="fill:white;stroke:black;" />
  <text x="71" y="101" fill="black" font-family="sans-serif"
font-size="16">x, y</text>
  <text x="126" y="61" fill="black" font-family="sans-serif"
font-size="16">c</text>
  <text x="121" y="96" fill="black" font-family="sans-serif"
font-size="16">&#x3B8;</text>
  <text x="175" y="27" fill="black" font-family="sans-serif"
font-size="16">x<tspan font-size="11" baseline-shift="sub">1</tspan>, y<tspan font-size="11"
baseline-shift="sub">1</tspan></text>
</svg>
</p>
</body>
</html>

```

The preceding code creates the notation and diagram in Figure A-2.

SVG and MathML embedded in an HTML5 page

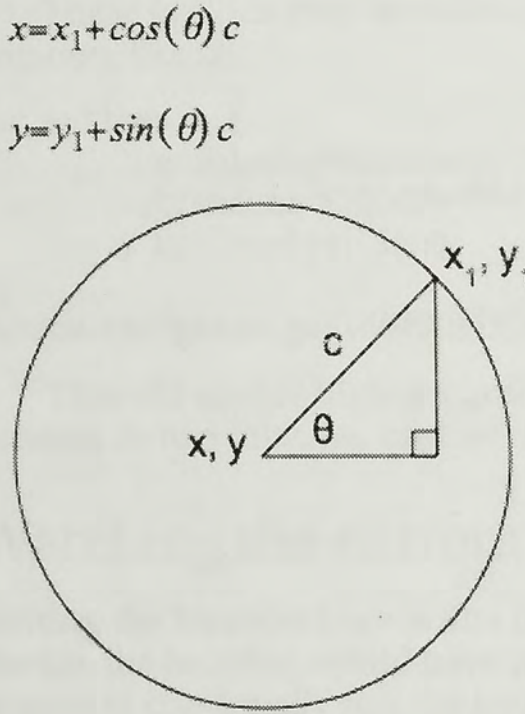


Figure A-2. A diagram created from markup using HTML, MathML, and SVG

Client-side storage

Imagine a web application that could save data the user had worked on in a client-side database and then sync that with a server-based database when the user connected online. Such offline functionality would be tremendously useful for improving the latency of the application, since the user's data would not need to be sent back and forth over the network as frequently, and it would also help in situations where connectivity may be spotty, such as in a mobile environment.

Web storage

Cookies have long been the method for storing data on the client-side browser. A problem with cookies has been that they are small, allowing only 4 kilobytes of storage each, which is minuscule by the standard of today's data-rich web pages/applications. In response to this, a new generation of client-side storage solutions have emerged. The most stable solution, and the one that can be seen as a replacement for cookies, is the Web Storage API,² which allows up to 5 megabytes of storage. Web storage is actually broken into two options, the localStorage object and sessionStorage object, both of which are properties of the window object. The difference between the two is that data stored in localStorage is persistent, while data stored in sessionStorage is lost when the browser session ends (such as when quitting the browser), but otherwise they are used in the same manner. Each is just a series of key/value pairs, so a key is set with some data, and then the key is used to retrieve the data later.

Using web storage

Using web storage is really quite straightforward. To add data to the storage, use either of the following syntaxes:

```

window.localStorage.setItem("key", "value");
window.localStorage["key"] = "value";

```

In this code, the "key" and "value" can be any string of text. To retrieve the data from the storage, use either of the following:

```

var val = window.localStorage.getItem("key");
var val = window.localStorage["key"];

```

To remove data, either remove a specific key or clear the whole storage:

```

window.localStorage.removeItem("key");
window.localStorage.clear();

```

WEB STORAGE EXAMPLE

Using the contenteditable attribute, you can create a simple text editor that saves changes on the client. For this example, create a new HTML file named edit.html and fill it with the following code:

```

<!DOCTYPE html>

```

² See <http://dev.w3.org/html5/webstorage/>.


```

<html>
  <head>
    <meta charset=utf-8 />
    <title>Contenteditable and localStorage demo</title>
    <script type="text/javascript" src="js/script.js"></script>
  </head>
  <body>
    <section id="editable">This text may be edited and the changes will
    be saved locally.</section>
    <button id="startEditBtn">Turn editing on</button>
    <button id="stopEditBtn">Turn editing off and save changes</button>
    <button id="clearBtn">Clear changes!</button>
  </body>
</html>

```

Now create a new JavaScript file named `script.js` and place it in a directory named `js` that is in the same location as `edit.html`. Fill it with the following script:

```

var editable; // variable for editable area

// initialize the variables and add event handlers
function init()
{
    editable = document.getElementById('editable');
    var startEditBtn = document.getElementById('startEditBtn');
    var stopEditBtn = document.getElementById('stopEditBtn');
    var clearBtn = document.getElementById('clearBtn');

    startEditBtn.onmousedown = startEdit;
    stopEditBtn.onmousedown = stopEdit;
    clearBtn.onmousedown = clear;

    // update text with data in local storage
    if (localStorage.getItem("savedtext")) editable.innerHTML =
    localStorage.getItem("savedtext");
}

function startEdit()
{
    // add the contenteditable attribute
    editable.setAttribute("contenteditable", true);
}

function stopEdit()
{
    // disable the contenteditable attribute
    editable.setAttribute("contenteditable", false);
    // save the text
    localStorage.setItem("savedtext", editable.innerHTML);
}

function clear()
{

```

```

// clear the local storage
localStorage.clear();
// reload the page
window.location.href = "";
}
window.onload = init;

```

Open the HTML page in a web browser, and you will be able to turn on editing (which adds the `contenteditable` attribute), save the edits, and see those edits stick because they will be stored in the local storage (Figure A-3).

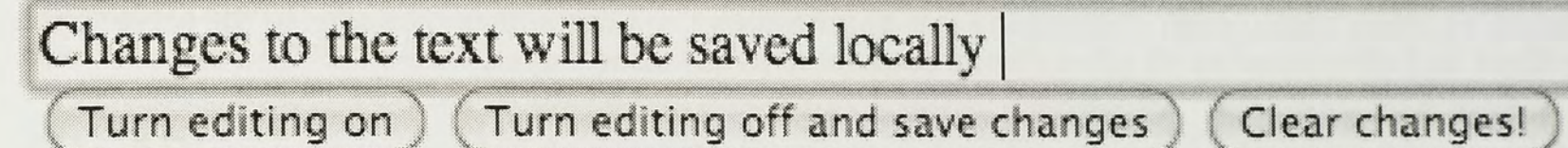


Figure A-3. A simple application using local storage

Other storage options

Local storage is easy to use, but with that ease comes limits on its capabilities. It's really not comparable to a database you would find on the back-end web server, which likely describes the relationship between the stored data and provides methods for ensuring data integrity. Since web technologies are moving toward enabling the creation of web applications, having a fully capable database on the client end is a desirable option. One such option is Web SQL, which essentially embeds an SQLite³ database into the web browser. This means **Structured Query Language** (SQL) commands can be used directly from JavaScript. Pretty cool! Unfortunately, the future of Web SQL has darkened considerably, because disagreements over standardizing the use of SQLite as the embedded database has led to support being dropped for the initiative by the W3C. Because of this, Mozilla has said it will drop support in Firefox, which means support is spotty and not reliable going forward. Too bad.

Another option, which is currently supported only in Firefox but has planned support from other major web browsers, is the Indexed Database API,⁴ also known as IndexedDB. This database solution stores key/value pairs, like web storage, but includes more sophisticated features, such as transactions for ensuring data is successfully committed to the database, which helps guarantee data integrity. IndexedDB is not as sophisticated as Web SQL (it's not a relational database), but it is more capable than web storage and is looking like it will be the option to use in the future for handling client-side data storage that is more complex than what web storage will accommodate.

Web workers

Web workers are making computationally intensive tasks on the Web a little less painful. JavaScript is a single-threaded language, meaning a script that takes a lot of processing power could completely paralyze any user-interactive scripts that may be running. Using a web worker, a new thread can be spawned that runs a script without interrupting the processing of UI interactions or other events in the main script. Web workers come in two flavors: dedicated workers and shared workers. Shared workers

³ See <http://sqlite.org>.

⁴ See www.w3.org/TR/IndexedDB/.

are more powerful than dedicated workers because they can communicate with multiple scripts, while a dedicated worker responds only to the script that spawned it in the first place.

Web Sockets API

The Web Sockets API⁵ is a specification that defines a protocol for providing two-way communication with a remote host. The Web's roots have traditionally been essentially one-way. A server sends a page to a client web browser, and then nothing happens between the two until the user clicks a link and requests another page. What a web socket provides is an open connection over which data can be sent from the client to the server at any time after the page has loaded, and vice versa. This could be used, for example, to create multiplayer online games or applications, because data can be sent to the server from one client and distributed to all other clients connected to the same server.

Video conferencing and peer-to-peer communication

A project is underway to create a specification for video conferencing between two web browsers. This is a major area of difference between the W3C HTML5 and WHATWG HTML specifications, because it is included in the WHATWG version but omitted from the W3C specification. Instead, the W3C has a separate specification named "WebRTC 1.0: Web Real-time Communication Between Browsers."⁶ Since both specifications are in draft status, it is not inconceivable that the version included in the WHATWG HTML draft may well be spun off into a separate specification in the future, as has happened at the W3C.

Anyway, administration issues aside, the actual technology for enabling video conferencing requires that two separate web browsers gather video and audio and stream it over a peer-to-peer connection to each other. Specifically, the following steps need to happen:

1. Gain access to a webcam or other video/audio input device.
2. Record the video/audio locally so that it can be streamed to a remote web browser.
3. Connect and send the video/audio to a remote web browser.
4. Display a video/audio stream in a video or audio element on the local and remote web browsers.

An API called the Stream API, which defines an interface called `MediaStream`, would be used with JavaScript to handle parsing and displaying of the streaming media. In terms of sending the media stream, another API, called the Peer-to-peer Connections API, would be used. This API describes a `PeerConnection` JavaScript interface that defines methods for connecting and sending a media stream to a remote peer.

⁵ See <http://dev.w3.org/html5/websockets/>.

⁶ See <http://dev.w3.org/2011/webrtc/editor/webrtc.html>.

WAI-ARIA

WAI-ARIA,⁷ the Accessible Rich Internet Applications specification (the WAI stands for Web Accessibility Initiative), aims to provide a syntax for making modern dynamic web applications accessible to people with disabilities. WAI-ARIA uses attributes to mark up user interaction of the page's content and describe how page elements relate to each other. WAI-ARIA defines a role attribute with a large set of values for describing how a web feature is presented and how the page is structured. There are also a large number of "aria-*" prefixed attributes for describing the state of web page features. These attributes can be used to annotate, for example, whether a menu has a submenu or what actions a dragged object can perform when being dropped on a target. The WAI-ARIA specification is a large specification in its own right; for further information on it, visit the WAI's WAI-ARIA overview page: www.w3.org/WAI/intro/aria.

File API

There are three specifications under development related to reading, browsing, and writing to files on the file system of a client machine. The main one is the File API,⁸ which includes interfaces called `File`, `FileList`, `FileReader`, and `FileError` that define methods that, for instance, can be used to read the name and last modification date of files or groups of files. The specification also defines a `Blob` for interfacing with raw binary data, which may be inspected for size and type and sliced into chunks. The File API deals with files that could appear in the web browser through a web form's file input type or even through dragging and dropping a file from the user's system to the web browser window. Extending the File API are the `Directories and System`⁹ and `Writer`¹⁰ APIs. `Directories and Systems` is what describes methods of interacting directly with the user's local file system. Obviously, this has security implications, so the file system exposed is sandboxed so web applications don't have unrestrained power to intrude into a user's computer. The `Writer` API does what you would expect; it defines how files or blobs of raw data can be written to the file system. It also defines a `FileWriterSync` interface for working with writing files in conjunction with the Web Workers API.

Useful web resources

The following websites are resources you may find useful when developing with HTML5:

- *W3C's Working Draft HTML5 specification*: <http://w3.org/TR/html5/>
- *WHATWG "living" HTML specification*: www.whatwg.org/specs/web-apps/current-work/
- *Html5.org*: Includes a HTML5 validator and a tracker for changes to the WHATWG specification: <http://html5.org>
- *Html5rocks.com*: Includes an online code editor playground and slide presentation made entirely with HTML5 technologies: <http://html5rocks.com>

⁷ See www.w3.org/TR/wai-aria/.

⁸ See www.w3.org/TR/FileAPI/.

⁹ See www.w3.org/TR/file-system-api/.

¹⁰ See www.w3.org/TR/file-writer-api/.

- *Html5doctor.com*: Contains informative articles about HTML5 as well as a comprehensive element reference; <http://html5doctor.com>
- *Caniuse.com*: Compatibility tables for HTML5, CSS3, and related technologies: <http://caniuse.com>
- *Html5test.com*: Browser score for HTML5 and related feature support: <http://html5test.com>
- *CSS3 Selectors Test*: Browser test for support of a wide range of CSS selectors: <http://www.css3.info/selectors-test/>
- *Mobilehtml5.org*: HTML5 feature compatibility tables for mobile and tablet browsers: <http://mobilehtml5.org>
- *HTML5boilerplate.com*: Starting template for HTML5 pages: <http://html5boilerplate.com>
- *Modernizr*: JavaScript library for testing browser support of HTML5, CSS3, and related features: <http://modernizr.com/>
- *Google Chrome Frame*: Method of enabling modern web technology capabilities in older browsers: <http://code.google.com/chrome/chromeiframe/>
- *Html5pattern.com*: Regex patterns for client-side validation in web forms: <http://html5pattern.com>
- *Mozilla Developer Network (MDN)*: Great, easy-to-follow resource on HTML5 and other web technologies: <https://developer.mozilla.org/en/HTML/HTML5/>
- *Html5gallery.com*: A showcase of sites using HTML5 technologies: <http://html5gallery.com>
- *Mediaqueri.es*: A showcase of sites using media queries: <http://mediaqueri.es>

Index

■ A, B

Adobe Dreamweaver, 154
 Advanced audio coding (ACC), 171
 Ajax, 1
Android, 269
 Application programming interface (API), 13
 Audio and video encoding, 173

■ C

Chrome, 229
 Client-side storage, 281
 contenteditable attribute, 281
 IndexedDB, 283
 local storage, 281, 283
 sessionStorage object, 281
 Web SQL, 283
 ColorZilla, 218
 Content model categories, 31
 content grouping elements, 47, 48
 document metadata and scripting elements, 36–37
 document sectioning elements, 45, 46
 embedded content elements, 71–73
 form elements, 68–70
 interactive elements, 73–74
 tabular data elements, 60–61
 text-level semantic elements, 55–59
 Cryptographic key generator, 135
 CSS2.1, 177–178
 CSS3
 :checked and :indeterminate selectors, 196
 :enabled and :disabled selectors, 195
 :focus selector, 195
 :in-range and :out-of-range selectors, 196
 :read-only and :read-write selectors, 196
 :required/:optional selectors, 196
 :valid/:invalid selectors, 196
 attribute selectors, 189–190

background color and images
 background-attachment property, 211
 background-clip property, 211–213
 background-color property, 210
 background-image property, 210
 background-origin property, 212, 213
 background-position property, 211
 background-repeat property, 210, 212, 213
 background-size property, 213–214
 basic settings, 210
 border-box, 211
 contain (L) and cover (R) values, 214
 content-box, 211
 padding-box, 211
 space and round properties, 210
 check box, 196
 class selectors, 183, 184
 class style rules, 185
 color
 additive color model, 217
 background and foreground colors, 218
 functional notation syntax, 219
 hex triplet, 217
 hexadecimal notation, 217, 218
 hue, saturation, lightness, 219–220
 opacity, 220, 221
 web color value calculation, 217–218
 combinators and complex selectors, 186–188
 combining selectors, 185–186
CSS box model
 block-level box, 205–207
 box layering setting, 209
 box position setting, 209
 box type setting, 207, 209
 inline-level box, 205–207
 line-level box, 205
 drop shadows, 216
 ID selectors, 183
location pseudoclass selectors, 190, 191

CSS3 (*cont.*)

- miscellaneous selectors, 203, 205
- modules, 178
- multiple backgrounds, 214–215
- pattern matching selectors
 - header and footer section, 200
 - nth-type selectors, 197
 - pattern formula, 199
 - tree-structural pseudoclass selectors, 197–198
 - unordered list, 199
 - zebra striping, 199
- pseudoelement selectors
 - ::before and ::after selectors, 201, 202
 - ::first-line and ::first-letter selectors, 201
 - generated and replaced content module, 201, 202
 - url() and attr() notation syntax, 201
- rounded corners, 215–216
- simple selectors**, 183, 184
- style rules, 181–183
- style sheet attachment
 - colors-deuteranopia.css style sheet, 180
 - colors-generic.css style sheet, 180
 - colors-protanopia.css style sheet, 180
 - firefox, 181
 - link element, 179
 - main.css style sheet, 180
 - meta element, 181
 - persistent, preferred and alternative style sheets, 181
 - rel attribute, 179
 - title attribute, 180
- type selectors, 183
- user action pseudoclass selectors, 192
- user interface states pseudoclass selectors, 194–195
- web typography, 221
 - multiple columns, 223
 - rules, 224
 - text effects, 223–224
 - web fonts, 221–222

D, E

- Doctype switching**, 10
- Document object model (DOM), 8

F

- File API, 285
- Firefox*, 229
- Form handler**, 107
- Form mastery**
 - accept-charset attribute, 109
 - acknowledgment page, 144, 145
 - action attribute, 109
 - autocomplete and novalidate attributes, 110
 - commentary gathering, 141–144
 - enctype attribute, 109
 - fieldsets and labels, 136–137
 - form controls**, 110
 - form handler**, 107
 - form input types, 111–112
 - form usability, 145–147
 - get and post methods, 107
 - input element attributes
 - autocomplete and autofocus, 126–127
 - data lists, 127–128
 - form input controls, 125
 - header and footer submit controls, 125
 - placeholder text, 126
 - readonly attribute, 126
 - input types
 - button, submit, reset, and image inputs, 117
 - check boxes, 115
 - color picker**, 118
 - date and time input, 118–121
 - e-mail, phone, and website URL inputs, 121
 - file input, 114–115
 - hidden input, 117
 - numerical inputs, 121
 - password input, 114
 - radio buttons, 116
 - search input, 121
 - text input, 113
 - menus
 - Boolean multiple attribute, 128
 - cryptographic key generator, 135
 - disabled and size attribute, 128
 - disabled, selected, value, and label attributes, 129
 - displaying progress, 132
 - gauge display, 132–134
 - list menu, 129
 - optgroup element, 130–131

- output element, 134
- select element, 128
- shorthand labels, 130
- text boxes, 131–132
- method attribute, 109
- name attribute, 110
- PHP code, 108
- shape-shifting, 111
- superglobal variables, 108
- target attribute, 110
- thank-you message, 144
- type attribute, 111
- user details gathering
 - datalist element, 140
 - e-mail address field, 140
 - generic text input field, 139
 - list attribute, 140
 - min and max attributes, 140
 - optional age field, 140
 - personal details fieldset, 140
 - placeholder text, 139
 - progress element, 139
 - regex pattern, 139
 - select element, 140
 - telephone and website input fields, 140
 - tip submission form, 137, 138
 - tipster's mailing address, 139
- validating and submitting forms
 - email input type, 122
 - image input control, 124
 - novalidate attribute, 122
 - pattern attribute, 123
 - regex**, 123
 - reset button, 124
 - submit buttons, 124
 - tel input type, 122
 - url input type, 122
 - validation error, 123
 - webforms2.js, 122
- web form, 108

G

- Geolocation API
 - clearWatch() method, 275
 - coords property, 276
 - error codes, 277
 - geolocation settings, 278
 - getCurrentPosition() method, 275–278
 - locatedFailed() function, 277

- locatedSuccess() function, 276, 277
- location query, 276
- satellite-based GPS, 275
- timestamp property, 276
- watchPosition() method, 275, 278
- Google Chrome, 255

H

- H.264 codecs, 164
- HTML5
 - Ajax
 - ajax/js directory, 234–235
 - event handling function, 234
 - hashtag, 237
 - init() function, 237
 - onpopstate event, 237
 - open index.html, 236
 - open() method, 234
 - prevButtonDown() and nextButtonDown() methods, 236, 237
 - pushState() method, 236, 237
 - replaceState() method, 236
 - var keyword, 234
 - XMLHttpRequest, 234
 - anatomy, 4
 - API, 13
 - attributes, 6–7
 - backward-compatibility, 8
 - browser layout engines and developer tools, 20
 - character encoding**, 11
 - content grouping elements**, 23, 46
 - blockquote, 50
 - description lists, 53
 - division element, 54
 - figure and figcaption elements, 54
 - horizontal rule, 49
 - inevitable paragraph, 49
 - pre element, 49
 - unordered and ordered lists, 50–53
 - content model categories, 12, *See also* Content model categories
 - copyright and legal information, 12
 - CSS technologies, 272–273
 - doctype declaration, 9–11
 - doctype switching**, 10

HTML5 (*cont.*)**document metadata and scripting****elements**, 23

- base element, 39
- character encoding information, 35
- CSS stylesheets, 35
- link element, 40–42
- script and noscript element, 42–44
- style element, 42
- web page information, 37–39

document sectioning elements, 23, 44–46**drag-and-drop operations**

- dataTransfer object, 255
- dragEndHandler() function, 257
- dragenter and dragover events, 256
- draggable attribute, 253–255
- dragLeaveHandler() function, 257
- dragOverHandler() function, 257
- dragStartHandler() function, 255–256
- dropHandler() function, 257
- dropzone attribute, 258
- edit script.js, 254
- event handling functions, 255
- events, 253–254
- list sorting, 259–261
- preventDefault() method, 256
- setData() method, 256
- styles.css, 254

elements, 5, 6**embedded content elements**, 24, 70–73**error handling**, 9**form elements**, 23, 67–70**global attributes**, 25

- accessibility, 26–27
- contenteditable attribute, 29
- core functionality, 24
- custom data, 31
- draggable and dropzone attributes, 30
- hidden attribute, 29–30
- id and class attribute, 27–29
- metadata, 27
- spellcheck attribute, 29
- style attribute, 30
- text directionality, 30

header, nav, and footer, 12**id attribute**, 12**interactive elements**, 24, 73

- menu and command, 76
- summary and detail elements, 74–76

JavaScript

- alert() method, 232
- Chrome*, 229
- CSS style sheet, 228
- document object, 232
- DOM, 228
- events, 233
- Firefox*, 229
- history object, 232
- HTMLVideoElement, 231
- Internet Explorer*, 229
- jstemplate, 227
- location object, 232
- navigator object, 231
- Opera*, 229
- prototype chain, 231
- Safari*, 229–231
- screen object, 232
- script element, 227
- stop() and play() methods, 231
- text editor, 227
- MathML, 13
- microdata, 13, 270–271
- MIME types, 17
- mobile web, 263
- mobile web pages testing, 268–269
- obsolete elements, 14–16
- offline application cache, 269–270
- open web technologies, 4
- presentational markup, 13
- responsive design
 - media queries, 266–268
 - screen size, 264
 - viewport, 264–266
 - web design standard, 264
- root element**, 23, 33–34
- scripted 2D canvas API
 - animation, 251–253
 - arcTo() method, 246
 - beginPath() method, 245
 - bezierCurveTo() method, 246
 - canvas path drawing methods, 246
 - canvas state, 249
 - CanvasRenderingContext2DPrototype, 243
 - closePath() method, 245
 - fill() method, 245
 - getContext() method, 241
 - getElementById() method, 242
 - interactivity, 250–251
 - JavaScript console, 243

- lineCap and lineJoin properties, 246, 247
- lineTo() method, 246
- lineTo(x,y) method, 245
- moveTo(x,y) method, 245
- quadraticCurveTo() method, 246
- rectangle drawing, 244–245
- stroke() method, 245
- triangle drawing, 246
- trigonometry, 247–248
- webgl, 243

semantic Web, 12**SGML**, 13**SVG**, 13**tabular data elements**, 23

- Adobe Photoshop, 59
- caption element, 63
- colgroup and col element, 65–67
- flow content model category, 59
- pixel-precise website layouts, 59
- table basics, 61
- table headers, 62–63
- thead, tfoot, and tbody element, 64–65

text/html, 17**text-level semantic elements**, 23, 54–59**undo manager API**, 272**video controller**

- block-level element, 240
- CSS **sprites**, 238, 241
- fallback content, 239
- getElementById() method, 239
- image sprite, 238
- init() function, 239
- overflow property, 240
- pauseVideo() function, 240
- play() method, 238
- stop() method, 238

W3C, 2, 3**Web applications 1.0**, 3, 4**web browser support**, 18–20**WHATWG**, 2, 3**XHTML**, 2, 18**HTML5 outlines**, 79, 88, 89**I, J, K****Indexed Database API**, 283*Internet Explorer*, 229**iOS**, 269**L****Layout engine**, 20**M, N****Mathematical Markup Language (MathML)**, 13, 279–280**Media queries**, 266–268**Metadata**, 27**Microdata**, 270–271**Mobile web**, 263**Multimedia****audio**, 170–172**audio and video encoding**, 173**canvas element**, 174**captions track**, 172–173**embed element**, 154–155**iframe element****frame and frameset elements**, 157**horizontal and vertical scrollbars**, 158*inline frame*, 157**longdesc attribute**, 159**Netscape Navigator 2.0**, 157*restricted local access*, 160**sandbox attribute**, 159–161**scrollbars**, 158**seamless attribute**, 159, 160**src attribute**, 158, 160**srcdoc attribute**, 159–160**target attribute**, 161**image maps****Adobe Dreamweaver**, 154**Boolean ismap attribute**, 152**client-side image map**, 153**coords attribute**, 154**href attribute**, 153**server-side image map**, 152**shape attribute**, 153**usemap attribute**, 153**img element****alt attribute**, 150, 151*CSS sprites*, 152**image-handling capabilities**, 152**ismap and usemap attributes**, 152**Mosaic web browser**, 149**src and alt attributes**, 150**title attribute**, 151**WebP**, 150**width and height attributes**, 151