HTML5 IN ACTION

Rob Crowther
Joe Lennon
Ash Blue

MEAP

MANNING

MEAP Edition
Manning Early Access Program
HTML5 in Action version 2

For more information on this and other Manning titles go to
www.manning.com

Table of Contents

1

HTML5: From documents to applications

This chapter covers

- What is HTML5 and what's new in this version
- How you can use HTML5 right now
- Tools to make HTML5 development easier
- New HTML elements, and how to use them

HTML5 is one of the hottest topics in Web development right now, and with just cause. Not only is it the latest version of the markup language for the Web, it also defines a whole new standard for developing *applications* for the Web. Previous iterations of HTML (and its rigid XML-based sibling, XHTML) have very much been centered on the concept of HTML as a markup language for documents. HTML5 is the first version that embraces the Web as a platform for Web application development, defining not only a series of new elements that can be used to develop rich internet applications, but also a range of standard JavaScript APIs for browsers to implement natively. A good example of this is the new <video> element, which provides a means of playing video content in the browser, without the need for an additional plug-in. Not only does HTML5 provide you with a means of including the video content on the page, but it also defines a series of JavaScript APIs that allow you to control its playback. You can create games. Build mobile apps. And much, much more.

In this chapter, you will learn about all of the great new features introduced in HTML5. You'll then dive right in and discover how you can use it in your Web applications right away, meanwhile learning how to provide fallbacks or workarounds for those users with older or incompatible browsers. Finally, you'll learn about the new elements introduced in HTML5, particularly those that aim to improve document semantics. You'll also learn about how to

use ARIA roles, microdata and microformats to further enhance the semantics of your HTML pages. By the end of the first chapter, you should have a broad sense of what HTML5 has to offer, how to use it gracefully in your applications, and what each of the new elements actually does.

1.1 What's new in HTML5?

From a functionality perspective, HTML5 includes a huge number of improvements over HTML 4.01 and XHTML. Rather than focus solely on the markup language itself, the HTML5 specification also standardizes an array of JavaScript APIs, helping Web application developers build software that will work consistently across a variety of browsers and platforms.

In this section, you will discover some of the exciting new features HTML5 introduces. Firstly, you will learn about the improved document semantics that HTML5 enables via a series of new semantic elements. Second, you will be introduced to the new form input elements and attributes that allow users to easily enter different types of data. Next, you will briefly discover the new <canvas> and multimedia elements that make HTML5 a viable replacement for native animations, games and multimedia content. Finally, you will learn about various JavaScript APIs that allow you to build feature-rich client applications with localStorage, offline support, drag and drop and more. Many of the features covered in this section are already implemented in at least one of the most popular Web browsers, and you will be able to try them out for yourself over the course of reading this book.

1.1.1 Better document semantics

Not so long ago, Web developers used complex table structures to define layouts for their Web pages. HTML tables, however, were never designed to be used for layout purposes, and it was semantically incorrect to use them in this way (not to mention the performance implications for rendering layer after layer of nested tables). With the introduction of Cascading Style Sheets, developers turned to clever styling of elements like to structure a page's layout, ensuring that tables could be left for representing tabular data.

In turn, this led to pages that contained an abundance of elements, typically given an ID or class to indicate what section of the page it represented. Ian Hickson of Google, the editor of the HTML5 specification, did some analysis on which class names are used on the most Web pages. The results can be found at http://code.google.com/webstats/2005-12/classes.html and make for interesting reading.

HTML5 aims to improve the semantics of Web documents by introducing a series of new elements that can be directly used in place of the element. Rather than authors picking and choosing their own class or ID values to give to sections of the page, they will use a standard set of elements, making it easier for applications like Web spiders and search crawlers to understand the different parts of the page, allowing them to provide better search results. Interestingly, many of the top 20 class names in the aforementioned Google

study are catered to by these new elements. You will learn much more about the new semantic elements later in this chapter.

1.1.2 Improved web forms

It rarely receives acclaim, but the humble Web form has played a major role in the emergence of the Web as a platform for application development. Web applications chiefly rely on the ability to accept user input and store this data in a database – and without form elements, this would not be possible.

As HTML5 is highly focused on Web applications, there are many improvements in Web forms, all of which can be used without breaking compatibility with older Web browsers. The basic text field has been used far beyond its primitive capabilities, and HTML5 aims to ease the burden by offering a series of compatible types, each of which provides enhancements over the simple text field. Table 1.1 identifies the new input types that are available.

Table 1.1. The new form input types introduced in HTML5

color	date	datetime	email	localtime
month	number	phone	range	search
time	url	week		

You can use all of these new input types in your Web pages right now, as older browsers will fall back to a standard text input type where it finds a type it doesn't understand. Some of the new input types will allow browsers to provide standard widget controls for given types of form field. For example, the "date" type should display some form of popup date picker. In fact, this feature is already available in the Opera browser, albeit with a horrendously ugly control (see figure 1). The "color" type should offer a color picker, like you might find in a graphics manipulation application. Once again, browser adoption of this feature has been slow, although Opera have supported it since version 11. Later in this chapter you will learn about Modernizr, an HTML5 feature detection script. Using Modernizr, you will be able to detect if a browser supports a given input type, providing a fallback JavaScript-based widget if required.

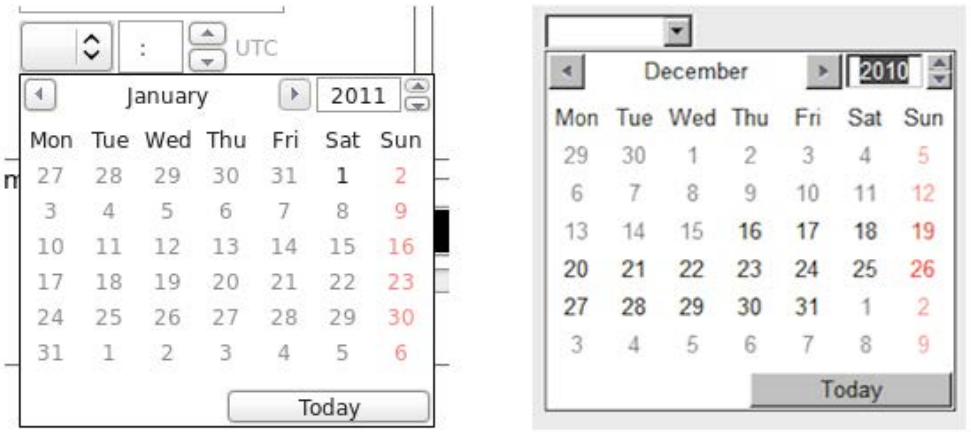

Figure 1.1. The HTML5 date widget in Opera running on Linux (left) and its ugly Mac OS X version (right)

In addition to new form field types, HTML5 also introduces a series of new common attributes that allow you to alter the behavior of a given field. For example, the new "placeholder" attribute can be used to provide a piece of text that should be displayed in a field when it is empty and does not have focus (see figure 2). This text is typically gray, and will be removed when the field is not empty or has focus.

Figure 1.2. HTML5 placeholder attribute in Opera

Table 1.2 is a complete list of the new attributes. You will learn about each of these attributes in detail, complete with usage examples, in Chapter 2, "HTML5 Forms".

Table 1.2. New input element attributes introduced in HTML5

autocomplete	autofocus	list	max	min
multiple	pattern	placeholder	required	step

Finally, HTML5 also introduces validation for form fields that does not require JavaScript. For example, if you have an email field, the browser will validate that the text entered by the user is a valid e-mail address. Of course, although this is not JavaScript validation, it is still client-side, and should never be trusted – you must still validate all your form submissions

on the server-side. You will learn about form validation and all of the new Web form features in great detail in Chapter 2, “HTML5 Forms”.

1.1.3 A new canvas for the Web

HTML provides a ton of different elements to allow you to present information on a Web page. These can be styled in many different ways, and you can use JavaScript to animate them and apply dynamic effects. If you are comfortable with complex JavaScript code (and are comfortable that your users will have a high-performance browser), you can even do pretty amazing things with simple HTML and JavaScript.

The problem is, there are many things that designers and developers may want to implement that HTML just doesn’t cater to. What if you want to insert a circle, square or other shape? What if you want to display an image and dynamically alter it based on user selections, on-the-fly? Of course, you could use static images or a server-side solution, but these are less than ideal in reality. The only viable solution in the past was to use a third-party plugin such as Adobe Flash.

Thankfully, HTML5 introduces the <canvas> element and a series of related drawing APIs that will allow you to do some amazing things without requiring the user to have a particular plugin installed. The <canvas> element is very well described by its name – it quite simply is a canvas for your Web pages. In figure 1.3 is a screenshot of a Breakout-style game, that was created entirely in HTML5 and JavaScript, with the game’s visuals output on a <canvas> element. Pretty neat, huh? Even neater is that you will learn how to build this game yourself in Chapter 6, “2D Canvas”.

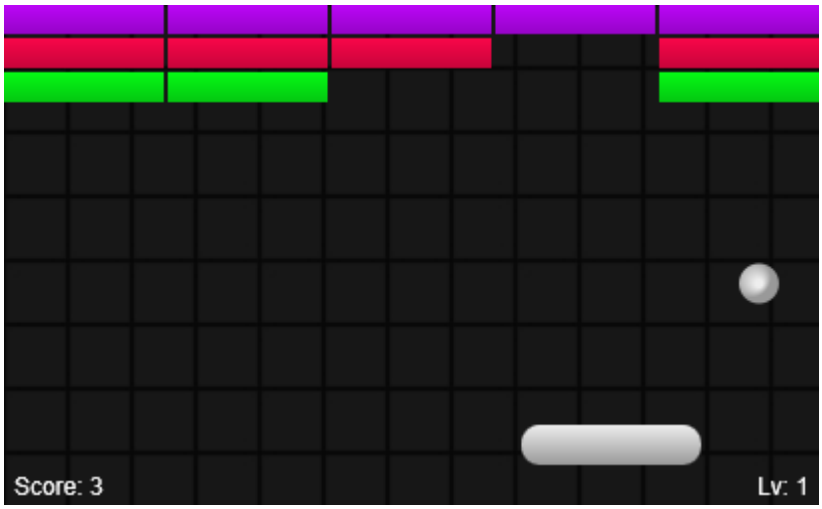

Figure 1.3. <canvas> allows developers to create some amazing things. You will learn how to build this Breakout-style game in Chapter 6, “2D Canvas”

Chapter 7 follows on from this by describing how to build a game using Scalable Vector Graphics (SVG) and JavaScript, illustrating that the Web is on the path to being a viable platform for the development of games. Chapter 8 takes things a step further by introducing the 3D canvas and OpenGL, showing you how to create a simple 3D world that you can navigate around using your browser.

Canvas is an exciting aspect of HTML5 – it opens up the Web to a huge number of possibilities. In chapter 6 you will learn how to use <canvas> in your own applications. With this information, and your imagination, the sky really is the limit when it comes to what you can achieve.

1.1.4 Native browser video and audio (with no added sugar)

One of the biggest growth areas on the Web in recent years has been streaming video, and to a lesser extent, audio. Both of these technologies are nothing new, and thanks to multimedia plugins like Shockwave, Flash, RealPlayer, Windows Media Player, QuickTime and others, the Web has been a viable medium for multimedia delivery for well over a decade.

With all of that said, restrictions on connection speeds and bandwidth severely limited the usefulness of such technology, restricting it to low quality streams and short clips. When MP3 came along and miniaturized the file size of high quality audio files, it changed the landscape of the music industry, causing the consumer market to shift towards digital as the industry stumbled to try and prevent a wave of mass piracy from crippling their profits.

Next, along came YouTube, which, coupled with the increasingly growing level of availability of high speed Internet access, led to a revolution in how people watched video. Flash-based video streaming took the Web by storm, and continues to lead the way today. What started off as a platform for self-publishing has moved to the streaming distribution of movies and TV shows via sites like Netflix and Hulu. With Internet access speeds climbing at a fast pace, streaming high definition (HD) content over the Web is steadily becoming the minimum acceptable standard.

All of this has been made possible thanks to browser plugins. Today, the majority of Web video is deployed in Flash video (FLV) format, an Adobe Flash container for various types of video codec. If the end user has a Flash plugin installed, they can view the video. Some have raised questions about the security and performance of Flash as a platform for video delivery, however, and are looking for alternative solutions. HTML5 provides such a solution through the new <video> and <audio> elements, which allow supported multimedia files to be played back natively by the browser (see figure 1.4 for a YouTube HTML5 video of Gmail Motion, a pretty funny April Fool's prank by Google), and controlled via a series of JavaScript APIs. This means that visitors can now view video content on your website, even if they don't have third-party plugins installed. This is particularly important if you want your content to be accessible by people using a device such as an iPhone or iPad, which do not support Flash at all.

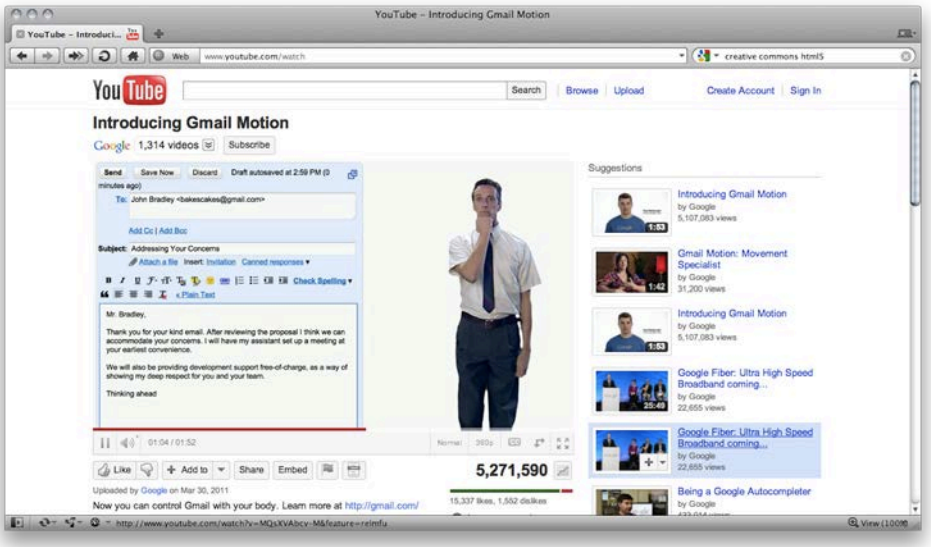

Figure 1.4. YouTube HTML5 video in action. The YouTube video in this screenshot does not use the Adobe Flash plug-in, but is fully implemented using the HTML5 <video> element and related APIs

HTML5 multimedia support is not limited to video content – the new <audio> element provides a similar feature set for audio playback. You will learn all about HTML5 video and audio in Chapter 8, "Video and audio".

1.1.5 Offline applications in an online world

One of the problems with Web applications is their dependence on being connected to the Internet at all times. While you can save Web pages for use offline, they are only usable if they are static and not reliant on a back-end database. Otherwise, you might be able to view the Web page, but as soon as you try to perform a query or update that requires server-side support, you'll run into trouble. In recent years, a number of solutions for working offline have emerged, including Google's Gears client-side database. Web applications that support Gears allow a user to use the application when offline, storing data locally, and when a connection becomes available, synchronize this data back to the main server. Of course, the problem here is that this offline support is completely dependent on the user having Google Gears installed, which the majority of Web users do not.

In HTML5, there are several new features that provide functionality allowing developers to enable access to their applications offline. First up is client-side data storage via the local storage and session storage APIs. This allows an application to store a reasonable amount of key-value pair data on the client-side, retrieving it as required. With the local storage API, the data will be kept on the client until it is either removed by the application or by the user.

The session storage API allows data to be kept only for as long as the browser session is kept open, it is automatically cleared when the session is closed. By default, browsers will allow an application to store up to 5 megabytes of data on the client-side, after which it will request permission from the user to grant access to more disk space.

In addition to this basic key-value storage mechanism, HTML5 also defines APIs for a more feature-rich client-side database. Originally, this was referred to as Web SQL, and was implemented by various browser vendors through the SQLite database system. The W3C did not want a single database system becoming the de-facto for all browsers, and preferred alternative implementations to be provided, and as a result, Web SQL was scrapped altogether. In its place, the W3C has provided IndexedDB, a document-oriented database API that behaves quite differently to traditional relational SQL-based databases. This new approach will allow an application developer to store complex data structures on the client-side, further enabling powerful offline Web applications.

Client-side storage solves the issue of accessing data while offline, but this is still not very useful unless the application itself is also accessible offline. It would be tedious if a user of your application needed to manually save your application for offline use, and it would become even more problematic when it comes to delivering updated versions of the application. Thankfully, HTML5 provides support for offline applications using an application cache manifest file and API. To use this functionality, your Web application must provide a manifest file, outlining the URIs (Uniform Resource Identifiers) that should (and should not) be cached by the browser for offline use. You can even define fallback files that should be served up in the case that a particular file is attempted to be accessed, enabling you to allow some parts of the application to be used offline, while requiring the user to be online for other parts. For example, you might have a word processing application, where the user can create and save documents offline, but in order to store these documents in the cloud or share them with others, they need to be working online. In this case, you would use localStorage to save the data on the client-side, using a cache manifest to ensure that the HTML and JavaScript files can be used offline. Any APIs that require online use could have a fallback that informs the user that they need to connect to the Internet in order to use that particular feature.

You will learn much more about local storage, session storage, IndexedDB and application cache in Chapter 5, “Offline and mobile applications”.

1.1.6 Look ma, no jQuery!

Because previous versions of HTML have lacked in any real JavaScript APIs for Web applications, developers have become increasingly dependent on external JavaScript libraries and frameworks such as jQuery, Prototype and so on. While these frameworks provide some excellent features, the increasing over-dependence on them is leading to new JavaScript developers becoming experts in JavaScript libraries rather than in JavaScript itself. This often results in highly inefficient coding practices and a lack of basic understanding in the principles of JavaScript on behalf of JavaScript developers.

The title of this section may be a little misleading – HTML5 is unlikely to eradicate the need and dependency on the likes of jQuery – but it does promise to at least make some of the features available using standardized, native and faster APIs. You can expect that libraries such as jQuery will be modified so that they use HTML5-enabled features by default, and fall back to other techniques for non-HTML5-friendly browsers.

The following HTML5 features are just some of those that would typically have required an external library in the past.

- Drag and drop
- History and state management
- hashchange event
- Basic data storage (HTML5 data-* attributes)
- Inline editing
- Touch events
- Undo changes

These features (and some other nice new features, such as the File API, and multithreaded Web development using Web Workers) will be covered in more detail in Chapter 4, “Creating desktop style Web applications”.

1.1.7 And if that isn’t enough…

At this point, we’re sure you’ll agree that HTML5 is bursting with new features and improvements for you to use in your own applications. It is important to remember, however, that the specification is far from finalized at this point, and it is likely to change quite substantially in the near future. HTML5 is regarded by many as not just a Web standard, but a term to describe a new paradigm for Web application development. The likes of Apple commonly refer to HTML5 in a way that describes any Web application that uses HTML5, CSS3, SVG, WebGL and other related technologies. In the final chapter of this book, Chapter 10, “The future of HTML5”, you will discover some features that may find their way into the specification before it is completed, such as the <device> element and the ping attribute, and also some related technologies that can be used in conjunction with HTML5 to provide an even better user experience. In the next section, you will learn how you can start using HTML5 in your Web applications right away, meanwhile ensuring that users with older, non-HTML5 browsers are not left behind.

1.2 Using HTML5 right now

Despite the HTML5 specification’s relative youth in the W3C standards process, Web browser support is improving rapidly. This means that there is no reason why you can’t start using HTML5 in your web applications right away. Before jumping in and adding HTML5 features to your heart’s content, however, there are a number of things you need to be aware of. This section will guide you through structuring HTML5 documents, providing support for older

versions of Internet Explorer, detecting browser support for particular features and introduces a starter template framework that allows you to develop cross-browser HTML5 applications quickly.

1.2.1 Basic structure of an HTML5 document

HTML5 documents are structured in the same way as older versions of HTML – you declare a doctype at the top of the document, and open and close the HTML document with a pair of <html> and </html> tags. Between these tags, you have two distinct sections, a <head> section where you place meta information and other non-content items such as stylesheets, and a <body> section where your page content should go. If you've written HTML pages or applications before, none of this will be new to you, but there are some subtle differences that you need to be aware of, which we will cover in this section. These are the HTML5 doctype declaration syntax, how to use the opening <html> element, and the shorter versions of the various elements you can use in the <head> section.

DOCTYPE DECLARATION

The first difference of note is in the doctype declaration on the first line of the document. In previous versions, the (X)HTML doctype contained long DTD (document type declaration) references that were difficult to remember and defined whether the document should be validated using the transitional, frameset or strict version of the HTML syntax. The following code is the declaration for an XHTML 1.0 Strict document:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
```

If, like us, you are not a fan of this declaration, you'll be delighted to learn that for HTML5 documents the following doctype is valid:

```
<!DOCTYPE html>
```

This is shorter and more memorable, but most importantly, it is enough to force standards mode in all modern browsers, even those that don't support the specific features of HTML5. This means that you can safely start using the HTML5 doctype in your pages right now, and be safe in the knowledge that updating the doctype will not wreak havoc in older browsers.

THE <HTML> ELEMENT

The doctype isn't where HTML5's preference for concise code ends. If you have created XHTML pages, you may be familiar with the xmlns attribute that was added to the <html> element, defining the XML namespace that the XHTML document fit into. As a result, the majority of web pages written in XHTML included the following opening <html> code:

```
<html xmlns="http://www.w3.org/1999/xhtml">
```

The xmlns attribute is actually invalid in HTML documents, and it was never really needed anyway. If an XHTML document omitted it, the value http://www.w3.org/1999/xhtml was used by default. Predictably enough, you should drop this attribute in HTML5 documents.

HTML 4 introduced the lang attribute for many elements, including <html>. The attribute's purpose was to be used by browsers to determine how to display parts of the page depending on the language specified. In practice, this was rarely used, but the lang attribute proved very useful for screen readers and other assistive technology. Take the word

"chat" for example, which in English means informal conversation, but in French means cat. This word is also pronounced differently based on what language it is spoken in. Screen readers such as JAWS (Job Access with Speech) will pay heed to the lang attribute of the <html> element when deciding how to pronounce words such as this. As a result, we highly recommend that you use the lang attribute in your HTML5 documents. The following is an example of the recommended opening <html> element for an English document:

```
<html lang="en">
```

The value of the lang attribute should equate to the two-letter ISO language code for the language the document is written in. See http://reference.sitepoint.com/html/lang-codes for a complete list of these codes.

THE <HEAD> SECTION

The <head> section of an HTML document is where you place meta information such as the character set encoding, the document's title, any stylesheets required for the document's styling, and depending on your preferences, any <script> tags for your JavaScript code. Nothing's really changed here, but HTML5 does allow you to be more concise. Take the following example of an XHTML <head> section:

```
<head>
    <title>My Web page</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <link rel="stylesheet" type="text/css" href="style.css" />
    <script type="text/javascript" src="script.js"></script>
</head>
```

This <head> section contains the page's title, character set, a link to an external CSS stylesheet and a reference to an external JavaScript source file. It is worth pointing out that all of this code is perfectly valid in HTML5, but it is a little more verbose than it need be.

HTML5 introduces a new charset attribute for the <meta> element which allows you to be far more concise when defining the character set your page uses. In addition, because HTML5 is not an XML language, you no longer need to close empty tags like <meta />. You can further reduce the character count by removing the type attributes from the lines that reference the external CSS and JavaScript files. All browsers will assume that a stylesheet is of type text/css unless otherwise specified, and similarly all <script> elements are assumed to be of type text/javascript by default. This leaves you with the equivalent HTML5 <head> section:

```
<head>
    <title>My Web page</title>
    <meta charset="utf-8">
    <link rel="stylesheet" href="style.css">
    <script src="script.js"></script>
</head>
```

More concise code is easier to read and maintain, and that is always a good thing. The code above will work fine in all modern Web browsers, so you don't need to worry about breaking compatibility in your existing documents. If you insist on worrying, fear not, the next section will introduce some problems that will surface with every Web developer's "favorite" browser.

1.2.2 Oops! IE did it again…

As you would expect, any new version of HTML will bring with it a series of new elements that can be used in your documents, and HTML5 is no different. You will learn about these new elements (and which ones have been dropped) in detail later in this chapter. For now, there's a little bit of housekeeping required in order to get all browsers to play friendly with the new elements.

In most modern browsers, all of the new elements will render fine, even in Internet Explorer (IE). However, when you try to style any of these elements, you'll find that IE decides to throw its toys out of the pram and sulk in the corner. Even if you have taken the noble action of de-supporting IE6 in your pages, you'll be shocked to read that IE7 and IE8 also react negatively to styles applied to HTML5-specific elements.

Fortunately, there are a few ways of alleviating this problem. The first option requires the use of JavaScript. If you want to use the element <header> on your page and need to apply some CSS styles to it, executing the following JavaScript code in the <head> section of your page will force Internet Explorer to apply the CSS rules to the tag, even if the version of IE used does not support a particular element natively:

```
document.createElement("header");
```

You will need to execute this for every HTML5-specific element you wish to use in your page. This will cause IE to render the style correctly, but the problem will persist if you attempt to print the page. Fortunately, a solution known as IE Print Protector can be used to fix the printing issue. Rather than reinvent the wheel, you should use Remy Sharp's HTML5 shiv script to do all this for you (or just use Modernizr, which we will cover in the next section). For more information on Remy's script, see http://remysharp.com/2009/01/07/html5-enabling-script/.

The problem with the previous solution is that it requires JavaScript to be enabled for it to work. If your page or application is heavily dependent on JavaScript, this might be an option for you, but if you need to support non-JavaScript visitors, you'll need an alternative solution.

The first non-JavaScript method is pretty ugly, to put it lightly. Basically, it involves serving the new HTML5 elements to anyone using a supporting browser, and serving good ol' elements to IE. This can be achieved using conditional comments, which will only be parsed by Internet Explorer, and ignored by all other browsers. By giving the HTML5 and element the same class, you will only need one CSS rule, which will apply to both approaches. Let's take a look at this (scary looking) markup.

Listing 1.1. Using conditional comments to support older versions of IE

```
<!--[if lt IE 9]><div class="header"><![endif]-->
<!--[if (gte IE 9)|(!IE)]><!--><header class="header"><!--<![endif]-->  #1
    <h1>This is my header</h1>
<!--[if lt IE 9]></div><![endif]-->
<!--[if (gte IE 9)|(!IE)]><!--></header><!--<![endif]-->
```

Believe it or not, that code is perfectly valid HTML5, and will pass the HTML5 validator at http://html5.validator.nu with flying colors. First, the code checks if the visitor is using some IE browser less than version 9, and if this evaluates to true, a element is used. Next, the code checks if the user has IE 9 or later, or is not using IE at all. If this is the case, the browser renders an HTML5 <header> element instead. At this point, you might be thinking – "wait a minute, I thought other browsers don't read conditional comments? How do they know to use <header>?" Notice how after the opening conditional comment tag we have an additional comment tag that immediately closes? #1 IE sees this as a nested comment and will read the code inside, but other browsers will close the conditional comment altogether, allowing the inner section to be parsed (and validated) correctly. This is repeated after the use of the <header> element to allow for the closing conditional comment tag.

A far less verbose option for enabling HTML5 elements to be styled in IE is to not use HTML5 at all, but rather opt for its less-popular sibling, XHTML5. The benefit of this approach is that all the new features of HTML5 are available to you, while maintaining backward compatibility for styling new semantic elements in Internet Explorer. The primary drawback is that you'll need to adhere to a strict standard of markup if you want your XHTML5 document to validate. In addition, you'll need to do some trickery on the server-side to send different MIME types to Internet Explorer and other browsers, which is less than ideal. For an excellent overview of this technique, see Eric Klingen's post on the subject at http://www.debeterevormgever.nl/www/html5-ie-without-javascript/.

1.2.3 Feature detection with Modernizr

One of the problems with using HTML5 right now is the lack of consistent browser support for the various features defined in the specification. For example, the new autofocus attribute for input elements works in Firefox 4 but not in Firefox 3.6. Safari 4 did not have support for Web Sockets, these were introduced in Safari 5. With the ever-expanding set of features in HTML5, and the ever-changing state of browser support amongst the major vendors, it would be exhausting trying to maintain a list of which browser supports which feature.

You can use JavaScript to easily detect if the visitor's browser supports a particular feature. For example, to check if they have support for offline applications, you would use the following code

```
!!window.applicationCache
```

This statement will evaluate to true if the HTML5 application cache is supported, or false if it is not. Unfortunately, not every HTML5 feature is detected in the same way. Local storage is also implemented as a property of the window object. As such, you might expect the following to work:

```
!!window.localStorage
```

This will work in many places, but if you try to use it in a debugging tool like Firebug, it will raise a security exception. Instead, you can consistently use the following statement.

```
'localStorage' in window
```

To detect some features, you have to go to much more trouble than the above approach. Let's take the canvas element as an example.

```
!!document.createElement('canvas').getContext
```

This code basically creates a dummy canvas element and calls the getContext method on it. The double-negative prefix on this statement will force the result of this expression to evaluate to either true or false, in this case informing you of whether or not the browser supports the canvas element. As a final example, let's look at how you would detect one of the new HTML5 form input element types, in this case the date type:

```
var el = document.createElement('input');
el.setAttribute('type', 'date');
el.type !== 'text';
```

Pretty ghastly stuff, huh? Of course, you could wrap this in a function to make it re-usable, but writing functions for each and every HTML5 feature would be painstaking. Thankfully, there is a JavaScript library named Modernizr that does all this for you.

To use Modernizr, grab the minified JavaScript source file for the library from http://www.modernizr.com, and include it in your HTML document by adding it to the `<head>` section, for example:

```
<script src="modernizr-1.7.min.js"></script>
```

You'll also need to add a `class` attribute to your document's `<html>` element, with the value `no-js`, as follows:

```
<html lang="en" class="no-js">
```

You can now use Modernizr to detect support for various HTML5 features. Let's see how you would use it to detect the four features we detected earlier in this section.

```
Modernizr.applicationcache //true if offline apps supported
Modernizr.localstorage //true if local storage supported
Modernizr.canvas //true if canvas supported
Modernizr.inputtypes.date //true if date input type supported
```

We're sure you'll agree that this is much easier to remember and read. Modernizr also adds a host of CSS classes to the `<html>` element of your document to indicate if a particular feature is available in the visitor's browser. This allows you to serve up different styles to users based on whether their browser supports a given feature. For further information on the Modernizr library, visit the project's website at http://www.modernizr.com.

1.2.4 HTML5 Boilerplate

If you are building an HTML5 application from scratch, there is quite a lot to watch out for. Ensuring your app is cross-browser compatible, supporting caching in an efficient manner, optimizing for mobile browsers, performance profiling, unit testing, writing printer-friendly styles – these are just a sample of the various complexities that come with the territory when building modern Web applications.

Rather than learning about and catering to all of these issues individually, wouldn't it be nice if you could get up and running quickly using a template that takes care of all of this for you? This is exactly what the HTML5 boilerplate does. The following is just a snippet of the features the boilerplate includes:

- Modernizr
- jQuery (hot-links to a Google-hosted file for performance, with a local fallback)

- Optimized code for including Google Analytics
- Conditional comments to allow for Internet Explorer-specific styling
- CSS reset, printer-friendly styles
- Google-friendly robots.txt file
- .htaccess file jam packed with site optimization goodness

We highly recommend using HTML5 Boilerplate as a starting point for all of your HTML5 applications. As the creators of the boilerplate point out, it is delete-key friendly – so if you don't want to include anything that comes as part of the boilerplate, you can simply remove it. The latest version of the project also supports custom builds, allowing you to include only those parts that you really need. For further information and to download the HTML5 Boilerplate, visit the project's website at http://html5boilerplate.com.

Now, having covered the various aspects of HTML5 that you can add to your existing Web pages and applications, let's explore the various tools that every HTML5 developer should have installed on their system.

1.3 An HTML5 developer's toolbox

When developing HTML5 applications, your development environment will primarily consist of a text editor or integrated development environment (IDE), and a Web browser. Every Web developer should at least have a copy of the latest versions of the major browsers:

- Apple Safari
- Google Chrome
- Microsoft Internet Explorer
- Mozilla Firefox
- Opera

Ideally, you'll have a virtualization environment available to you that will allow you to test several versions of each browser on the same machine. For example, you can use applications such as Oracle VirtualBox, VMware Fusion/Player/Workstation or Parallels Desktop to create multiple virtual machines with different versions of browsers installed in various different operating system environments. This is extremely convenient for testing how your application will work in a wide variety of different browser and system configurations.

In addition to installing the major Web browsers, you should also ensure that you have the relevant tools available to make your life easier. All of the major browsers now include a suite of Web developer tools, either as standard or separately via a browser plug-in. These tools are vital when it comes to testing, debugging and analyzing the performance of your Web pages and applications. The features provided by these tools include:

- Console output
- JavaScript debugging
- Element and property inspection
- Network activity and traffic analysis
- JavaScript performance profiling
- On-the-fly element styling and manipulation

In this section, you will learn about the developer tools available in each of the major Web browsers. You will discover how to access Safari and Chrome's Developer Tools, how to download and install Firebug and Web Developer plug-ins for Firefox, how to start the Dragonfly tool in the Opera browser, and finally where you can launch Developer Tools in the various versions of IE.

1.3.1 Safari and Chrome Developer Tools

Both Safari and Chrome include a set of developer tools out-of-the-box that provide all of the aforementioned features. In Chrome, you can launch these tools from the main menu, under Tools > Developer Tools. In Safari, the tools are hidden by default. From Safari's Preferences dialog, check the "Show Develop menu in menu bar" option in the Advanced section. You will then see an additional menu in Safari called "Develop". The "Show Web Inspector" menu item will open Safari's Developer Tools. See figure 1.4 for an example of Chrome's Developer Tools in action.

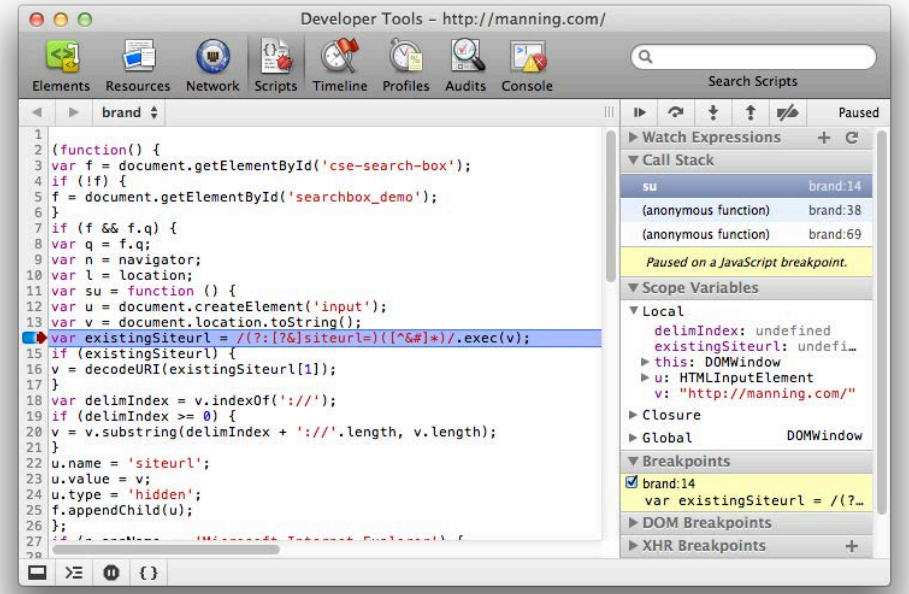

Figure 1.4. Google Chrome Developer Tools showing how you can set breakpoints, watch expressions and change variable values to perform JavaScript debugging directly in the browser

The Developer Tools included with Safari and Chrome are rich in features, including DOM live editing, full JavaScript debugging, network and JavaScript analysis tools, and the ability to view cookie and other data stored on the client-side. The tools can be docked to the bottom of the browser window or opened separately in their own application window. In addition, Safari's Develop menu makes it easy to enable and disable support for the likes of images, JavaScript, style sheets and cookies, allowing you to quickly test how your application works under those constraints.

1.3.2 Firebug and Web Developer plug-ins for Firefox

Although Firefox doesn't include a suite of developer tools by default other than a basic error log window, there are several plug-ins available that will add such features to Firefox. The primary developer plug-in for Firefox, Firebug, is actually maintained by Mozilla themselves, and includes most of the features you can find in Safari/Chrome Developer Tools, including console logging and output, live DOM, JavaScript and style editing, JavaScript debugging with support for breakpoints and watch expressions and network analysis tools. You can download Firebug at http://getfirebug.com/. Firebug can be seen in action in figure 1.5.

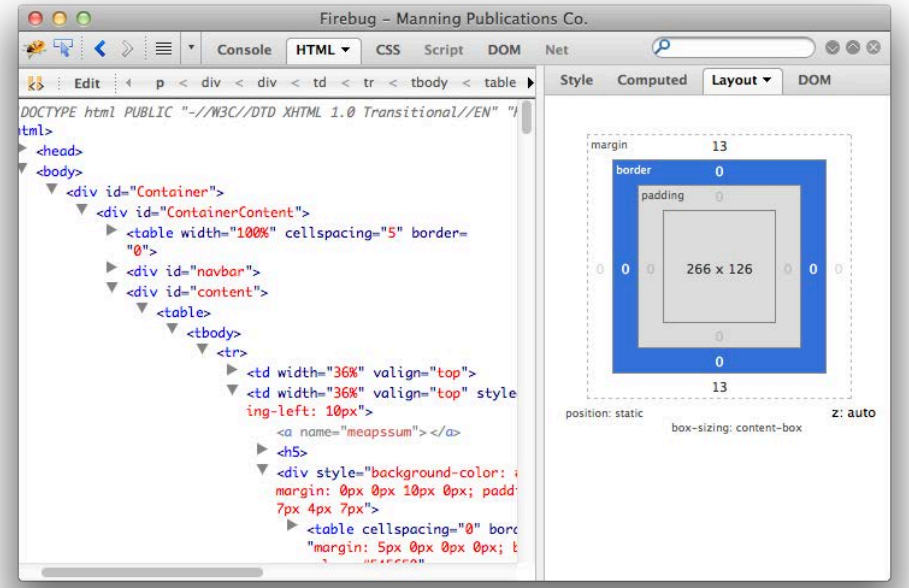

Figure 1.5. Firebug plugin for Mozilla Firefox, illustrating how you can view the HTML source and styling properties for a Web page, which is dynamically updated when the source is modified in JavaScript

Firebug does not include all the tools you might need, for example, it does not include features to view detailed information about cookies, or indeed the ability to easily switch on and off things like images, cookies and JavaScript. Fortunately, all of these features, and more besides, can be added with the Web Developer plug-in. You can download and find out more about the Web Developer plug-in on the Firefox Add-ons website at https://addons.mozilla.org/en-US/firefox/addon/web-developer/.

1.3.3 Opera Dragonfly

Like Chrome and Safari, the Opera browser also natively includes a suite of Web developer tools, called Dragonfly. You can launch Dragonfly in Opera from the “Tools > Advanced > Opera Dragonfly” menu. Dragonfly can be seen in action in figure 1.6.

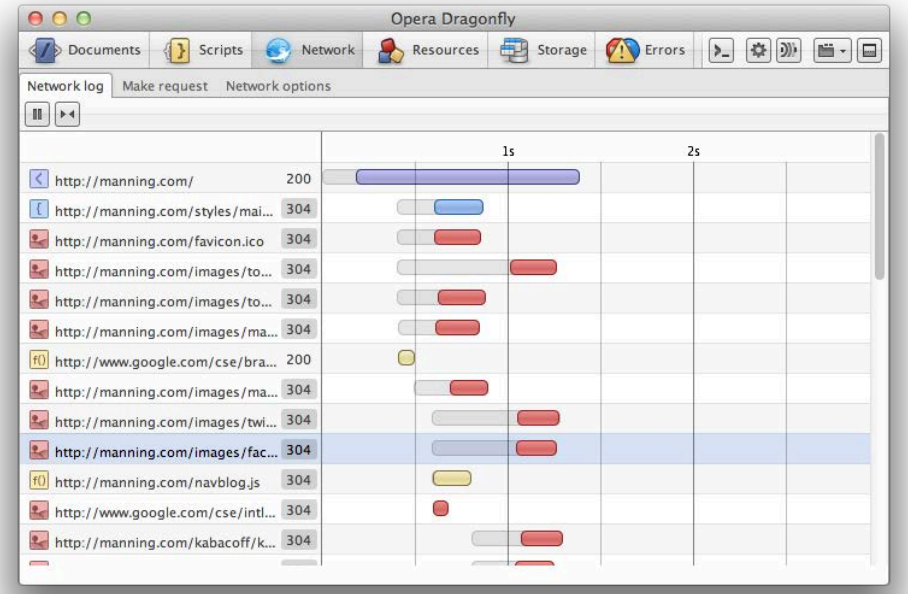

Figure 1.6. Opera Dragonfly displaying a visual representation of the time each resource on a page takes to load. Using this graph, you can easily identify the parts of your application that take the longest to load.

Dragonfly includes most of the features you'll find in Safari/Chrome Developer Tools, but with a few neat additions like the ability to take screenshots and pick colors from Web pages, without requiring additional plug-ins or tools to be installed.

1.3.4 Internet Explorer Developer Tools

As of version 8, Internet Explorer includes a set of web developer tools known simply as "Developer Tools". Unfortunately, these tools are quite basic and until IE9, did not include several features that can be found in the tools for competing browsers, such as network analysis and console logging. These missing features can be supplanted through the use of additional external tools, but you are probably just best to use an alternative browser as your standard development browser. If you want to get Developer Tools for IE6 or IE7, you can download it in the form of the "Developer Toolbar". Figure 1.7 shows a screenshot of IE8's Developer Tools.

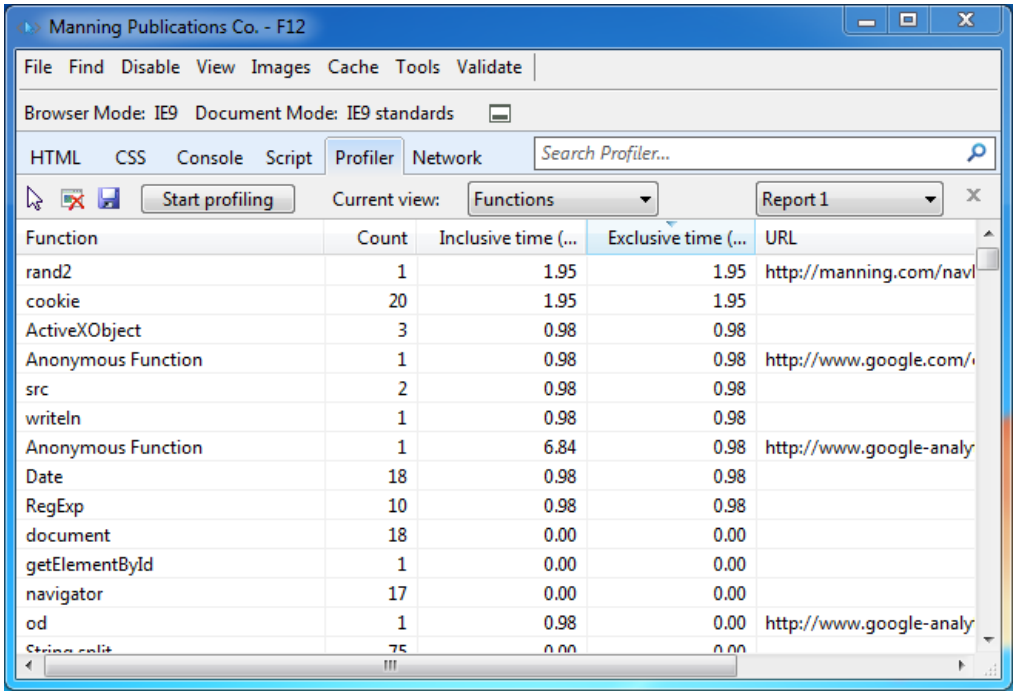

Figure 1.7. Microsoft Internet Explorer 8 Developer Tools showing how JavaScript profiling allows you to measure the time functions and blocks of code take to execute, allowing you to optimize your code for optimum performance.

Additional tools are available for Internet Explorer such as DebugBar, which adds some missing tools to older versions of IE, as well as some nice extras such as HTML validation, taking screenshots and a color picker. DebugBar can be downloaded from http://www.debugbar.com/.

Now that you're equipped with the tools you need to to start writing HTML Web applications, we'll help you put those tools and your new knowledge to use with the new elements introduced in HTML5.

1.4 Using HTML5's new elements

When any new version of HTML is released, it typically defines a series of new elements that developers can use in their Web pages. HTML5 is no different, and defines quite a large number of new elements, and deprecates or removes some others. In this section, you will discover the new semantic elements that are included in HTML5, and how to use them appropriately in your own markup. Next, you will learn about how using ARIA roles can enable assistive devices to better cope with modern, sophisticated Web applications. Following this, you will see how you can use microdata to further enhance the semantics of

your HTML documents, allowing search engines to make more sense of different aspects of your Web pages. Finally, you will find out what elements and attributes have been removed from the HTML language altogether in HTML5. Let's get started!

1.4.1 The new semantic elements

If you have read about HTML5 before you picked up this book, chances are you've heard plenty about the new semantic elements. Although they are relatively straightforward, there is a tendency for people to get overexcited about this new set of tags. They are quite important, particularly if you want search engines and assistive devices such as screen readers to understand your pages better, but from a functional perspective the new elements are really simple.

If you're expecting these new elements to do something magical in terms of how they look on your page, you're in for quite some disappointment. Using these new elements on your page is functionally equivalent to using a series of elements; they behave as block elements by default and can be styled as required using CSS. Their importance comes from the standard semantic meaning they have. If you think of a typical blog post, the Web page probably contains a series of sections. First, you have the site heading and navigation, maybe some side bar navigation, a main content area, and a footer area with further navigation links and perhaps some copyright and legal links. The following listing demonstrates how such a blog post might have been marked up in HTML 4 or XHTML.

Listing 1.2. HTML 4 markup for a blog post

```
<div class="header">
    <h1>My Site Name</h1>
    <h2>My Site Slogan</h2>
    <div class="nav">
        <ul><!-- Main Site Nav here --></ul>
    </div>
</div>

<div class="sidebar">
    <h3>Links Heading</h3>
    <ul><!-- Sidebar links --></ul>
</div>

<div class="main">
    <h4>Blog Post Title</h4>
    <div class="meta">
        Published by Joe on 01 May 2011 @ 12:30pm
    </div>
    <div class="post">
        <!-- Actual blog post -->
    </div>
</div>

<div class="footer">
    <ul><!-- Footer links --></ul>
    <!-- Copyright info -->
```

```
</div>
```

First off, there is nothing wrong with this code. In fact, it is perfectly valid HTML5 and you can absolutely continue to use elements with semantic class names if you wish. However, from a semantic point of view, there are a number of problems with the approach. First off, we are separating the various areas of the blog post using named classes. This is fine, but it is completely up to the author how the classes are named. My "header" might be your "heading"; I call the main section "main"; you might call it "body" or "article". In addition, some people may prefer to use IDs instead of classes. In short, there is no way for a search engine or other computer-controlled application to consistently determine what each section actually represents.

This is where the new semantic elements come into play. Rather than using classes and IDs for sections like headings, navigation, footers, and so on, you now use a standard HTML element. Let's look at these new elements now.

- **<header>** This element contains the title of a page or section. You can use this multiple times on the same page—that is, once for the page heading (logo, slogan) and again for the post heading (blog post title, permalink, meta information).
- **<hgroup>** You can use the <hgroup> element to group numbered heading (<h1> to <h6>) elements where you have more than one. In HTML5, the numbered heading elements are relevant to the section – you can have an <h1> in your site's main heading, and then a separate <h1> in your actual blog post. The heading structure doesn't apply to the entire page like it did in HTML 4.
- **<footer>** This usually appears at the bottom of a page or section, typically used for things like related posts or links, copyright information, meta data and more.
- **<nav>** This element represents a section of links that allow the user to navigate to other pages or parts of the same page. The specification dictates that only "major navigation blocks" should use the <nav> element – main site navigation, table of contents or side navigation bars are all usually considered "major navigation blocks".
- **<section>** This is used to define a section of a document or application – in a wiki article this might be a major section of the article, in an application it might be a popup form that allows you to add data. Sections can have their own headers, navigation and footers if required.
- **<article>** An article is deemed to be a self-contained publishable composition that can be redistributed on its own, for example as an entry in an RSS feed. Articles are typically items like blog posts, comments, forum posts, news entries, and so on. Like sections, articles can themselves contain headers, navigation and footer elements.
- **<aside>** The <aside> element is used to define a section of a page that is separate to the content area it is defined in. In a book or magazine, this might be represented as a sidebar – it contains information on the same topic but doesn't quite fit into the main article itself. For example, if you had a blog, you may have adverts displaying

alongside posts – these could be placed in an <aside> element.

- **<time>** Dates and times can be represented in a wide range of formats, languages and abbreviated styles. This makes it difficult to parse dates consistently. The <time> element allows you to continue to use the date/time format of your choice, meanwhile wrapping it in a standardized format that is easier for a computer to understand. The <time> element also offers a pubdate attribute which can be used to denote the publish date of an article such as a blog post.
- **<mark>** You can use the <mark> element to represent a part of text in your document that should be marked or highlighted. A common use for this would be to highlight search terms within a document.

The following listing illustrates how we might re-write the earlier HTML 4 example using HTML5 and some of the new elements.

Listing 1.3. HTML5 markup for a blog post

```
<header>
    <hgroup>
        <h1>My Site Name</h1>
        <h2>My Site Slogan</h2>
    </hgroup>
    <nav>
        <ul><!-- Main Site Nav Here --></ul>
    </nav>
</header>

<nav>
    <h1>Links Heading</h1>
    <ul><!-- Sidebar links --></ul>
</nav>

<section>
    <article>
        <header>
            <h1>Blog Post Title</h1>
            <div class="meta">
                Published by Joe on
                <time datetime="2011-05-01T12:30+00:00" pubdate>
                    01 May 2011 @ 12:30pm
                </time>
            </div>
        </header>
        <section>
            <!-- Actual blog post -->
        </section>
    </article>
</section>

<footer>
    <ul><!-- Footer Links -->
    <!-- Copyright info -->
```

```
</footer>
```

Next, you'll learn how you can improve the semantics of your page even further using ARIA roles, which greatly enhance the accessibility of your Web pages.

1.4.2 Enhanced accessibility with ARIA roles

When building Web applications, you must take care to ensure that your application is accessible to all users, including those who require assistive devices such as screen readers. Ensuring that your documents are accessible requires careful consideration when it comes to the semantic meaning of your markup. Using simple HTML markup this is relatively straightforward, and HTML5's new elements improve the semantics even further. However, when you move towards the area of Web applications, it becomes much more difficult to cater to assistive technologies. Because of the large amount of JavaScript code used to dynamically modify Web pages in modern Web apps, it may seem almost impossible to keep your application working in an accessible way. This is where the Web Accessibility Initiative (WAI) and Accessible Rich Internet Applications (ARIA) roles come into play.

The WAI-ARIA specification intends to improve Web applications by expanding on the accessibility information provided by the author of an HTML document. It requires that applications provide full keyboard support that can be implemented regardless of the device, making it easier to use on a smart phone, e-Book reader or Internet-ready television. Most importantly, perhaps, the specification focuses on improving the accessibility of dynamic content generated by scripts and providing interoperability for assistive technologies.

ARIA roles, relationships, states and properties allow you to define exactly how your Web application works in a way that an assistive device such as a screen reader can understand. The HTML5 specification explicitly states that you may use the ARIA role and aria-* attributes on HTML elements as described in the relevant ARIA specification. HTML5 also defines a set of default ARIA roles that apply to certain HTML elements – for example it is implied that a checkbox <input> element has an ARIA role of "checkbox", and you should not explicitly use the role or aria-* attributes in these cases.

There are also various HTML elements where the native semantics can be modified so that they behave differently. For example, you might have an <a> element which behaves like a button, perhaps submitting a form after performing some validation routine. The HTML5 specification defines a list of valid semantics for these elements, for example, when you use the <a> element to create a hyperlink, it assumes the "link" role by default, and if this is modified its role can only be changed to one of the following: "link", "button", "checkbox", "menuitem", "menuitemcheckbox", "menuitemradio", "tab" or "treeitem".

For a full list of both the default implied ARIA semantics and the restrictions on how you can modify the semantics of certain elements, see the WAI-ARIA section of the HTML5 specification at http://dev.w3.org/html5/spec/Overview.html#wai-aria.

In the next section, we'll look at some other concepts that are closely tied into HTML5, albeit not strictly defined in the specification itself: microdata and microformats.

1.4.3 More semantics with Microdata

When most people discuss HTML5, they are usually referring not just to the HTML5 specification itself, but also to a number of related specifications. The HTML microdata specification allows additional semantic information to be added to a Web page, allowing the likes of search engines and Web browsers to provide additional functionality to the user based on this data.

To use microdata, you need a vocabulary, which defines the semantics that are to be used. You can define your own vocabularies, but more likely you'll want to use generic versions, such as those provided by Google at http://www.data-vocabulary.org/, including Event, Organization, Person, Product, Review, Review-aggregate, Breadcrumb, Offer and Offer-aggregate. By using a generic vocabulary where possible, your microdata will be easier to digest for search engines and other applications, as they will also use this vocabulary. In a moment, we'll look at an example of defining an event using microdata. First, we need to look at the various HTML properties that allow us to use microdata on our pages.

- **itemscope** The itemscope attribute tells the parser that this element and everything contained inside it describes the entity being referenced. The value of this attribute is Boolean, and is usually omitted. When an element has the itemscope attribute, it also defines the itemtype attribute.
- **itemtype** This attribute defines the URL at which the vocabulary for the item being specified is found. For example, if you are using the Person microdata vocabulary provided by Google, the value of this attribute would be "http://data-vocabulary.org/Person". This attribute is defined on the same element as the itemscope attribute.
- **itemid** This attribute is a global unique identifier for the item. It is optional, but if used, it must be a valid URL.
- **itemprop** This attribute indicates that the content of the element contains the value of the specified property. For example, the following code `<span itemprop="name">Joe Lennon</span>` denotes that the name property of this item is "Joe Lennon". The itemprop attribute is the one that you will likely use the most when working with microdata.
- **itemref** If you want to specify microdata attributes on an element that is not contained inside an element that defines the itemscope attribute, you can use the itemref property to refer to that item instead. The value of this attribute must be an unordered set of unique space-separated tokens that contain the IDs of elements on the same HTML page. The itemref property must be put on the same element you place itemscope on.

Listing 1.4 illustrates microdata in action using an event item that adheres to Google's Event microdata vocabulary at http://data-vocabulary.org/Event. This code creates a snippet of HTML code for an event, with microdata properties defined that will allow a search engine to

provide better interpretation of the event, perhaps by showing the event date in a calendar, or location on a map.

Listing 1.4. Microdata in action

```
<div itemscope itemtype="http://data-vocabulary.org/Event">
    <a href="http://example.com/event/1" itemprop="url">
        <span itemprop="summary">John's 40th Birthday Party</span>
    </a>
    <span itemprop="description">To celebrate John turning 40,
    we're throwing a BBQ party in his honour this Friday evening
    at close of business. Please come and bring your friends and
    family!</span>

    Location:
    <span itemprop="location" itemscope itemtype="http://data-
vocabulary.org/Address">
        <span itemprop="street-address">500 Market Way</span>
        <span itemprop="locality">Ballincollig</span>
        <span itemprop="region">Cork</span>
    </span>

    Date and Time:
    <time itemprop="startDate" datetime="2011-05-06T18:00+00:00">
        Fri, May 6th @ 6pm
     </time>
</div>
```

Other means of annotating HTML syntax include the RDFa specification and using microformats like hCalendar, hCard and hProduct. All three approaches can be used to improve your site's search listings in Google. For more information on using these approaches, see http://www.google.com/support/webmasters/bin/topic.py?topic=21997. Now, let's wrap up this chapter by looking at the parts of HTML that have been removed in HTML5.

1.4.4 Out with the old…

A number of elements and attributes are absent from HTML5. They will typically continue to work in major browsers due to backward compatibility, but the specification requires that developers do not use them in their Web pages going forward. The list of removed elements appear in table 1.3.

Table 1.3. Elements no longer supported in HTML5

<acronym>	<applet>	<basefont>	<big>	
<dir>	<font>	<frame>	<frameset>	<isindex>
<noframes>	<strike>	<tt>	<u>	

In addition, a series of attributes are no longer allowed. Some are purely presentational attributes, and we recommend that you use CSS instead of these attributes, which will no longer validate in HTML5. Others are merely deemed to no longer be required. Note that some of the attributes listed in table 1.4 may still be perfectly valid on other elements (for example, the type attribute on the <input> element). Next to each attribute, we list in parentheses the elements to which it no longer applies.

Table 1.4. Attributes no longer supported in HTML5

abbr (<td>, <th>)	align (all presentational elements)
alink (<body>)	archive (<object>)
axis (<td>, <th>)	background (<body>)
bgcolor (<body>, <table>, <tr>, <td>, <th>)	border (<table>, <object>)
cellpadding (<table>)	cellspacing (<table>)
char (<col>, <colgroup>, <tbody>, <td>, <tfoot>, <th>, <thead>, <tr>)	charoff (<col>, <colgroup>, <tbody>, <td>, <tfoot>, <th>, <thead>, <tr>)
charset (<a>, <link>)	classid (<object>)
clear ( )	codebase (<object>)
codetype (<object>)	compact (<dl>, <menu>, ,)
coords (<a>)	declare (<object>)
frame (<table>)	frameborder (<iframe>)
height (<td>, <th>)	hspace (<img>, <object>)
link (<body>)	longdesc (<img>, <iframe>)
marginheight (<iframe>)	marginwidth (<iframe>)
name (<img>)	nohref (<area>)
noshade (<hr>)	nowrap (<td>, <th>)
profile (<head>)	rev (<a>, <link>)
rules (<table>)	scheme (<meta>)
scope (<td>)	scrolling (<iframe>)
shape (<a>)	size (<hr>)
standby (<object>)	target (<link>)

text (<body>)	type (, , <param>,)
valign (<col>, <colgroup>, <tbody>, <td>, <tfoot>, <th>, <thead>, <tr>)	valuetype (<param>)
version (<html>)	vlink (<body>)
vspace (<img>, <object>)	width (<col>, <colgroup>, <hr>, <pre>, <table>, <td>, <th>)

Alongside missing elements, there are also some elements whose behavior has changed in some shape or form between HTML 4 and HTML5. For detailed information on these changes, visit http://www.w3.org/TR/html5-diff/#changed-elements.

1.5 Summary

HTML5 is the most important revision of the HTML markup language since its inception in 1991. Although HTML started off as a relatively straightforward markup language, it has since become a platform for complex Web page design and Web application development, particularly when coupled with its close relations CSS and JavaScript. HTML5 is the first version of the language to acknowledge this and include a series of standard JavaScript APIs in the specification itself.

You should now be able to create HTML5 ready pages of your own, with the new doctype, improved semantics and techniques to ensure backward-compatibility with older Web browsers and in particular, versions 6, 7 and 8 of Internet Explorer. At this point, you are probably as sick of hearing about old versions of browsers as we are, and you want to get right into developing some cool applications. Fear not! Starting with the next chapter, you will be developing a sample app in each and every chapter. By the time you've finished the book, you'll be able to say you've built some awesome games, mobile apps and much more besides. And no, that's not a typo, we did say games, plural!

But first, in the next chapter, you will go beyond introductory concepts and gain a deep insight into the vast improvements in Web Forms that HTML5 has to offer, including the new input types that allow entry of a much wider variety of data types, new attributes such as autofocus and placeholder, and the out-of-the-box features that reduce the reliance on JavaScript for client-side data validation.

2

HTML5 Forms

This chapter covers

- New form input types
- New field attributes
- New HTML5 elements for data output
- CSS3 pseudo-classes and Constraints API
- Providing fallback for older browsers

Web forms are one of the most important aspects of HTML, but they haven't exactly evolved over the years – until now. Web forms allow users to enter data on a Web page, and then submit this data so that it is sent to the application or Website's server-side for further processing. Although Web forms can be as simple as a search box or a contact form, they typically play a very important role in gathering data from your application's users. Secure login, new user registration, credit card payment processing – all of these common and important forms are typical examples of what you can do with Web forms.

When HTML form elements were first introduced, they were perfect for adding basic dynamic functionality to Web pages such as contact forms or search boxes. As the Web has matured however, a need has emerged for a much richer set of form field types and widgets. Up until now, Web developers looked to third-party UI libraries to provide these components, with the likes of jQuery UI, ExtJS, YUI and Dojo all catering for this with rich suites of custom controls such as sliders, auto-complete combo-boxes, date pickers and more. As good as these frameworks are, they are all dependent on JavaScript to provide this enhanced functionality – which means in the majority of cases they won't work at all without JavaScript, and they may add a performance burden to sites that use them as they have to load a lot of additional CSS and JavaScript code.

In HTML5, this burden is reduced significantly with the introduction of thirteen new form input types, ranging from number spinners and sliders to date pickers and color pickers. In addition, HTML5 defines a series of new attributes that can be applied to input elements that will enhance the functionality of the form, including presentational attributes like placeholder and autofocus, as well as validation attributes such as required and pattern. You can even use a set of new CSS3 pseudo-classes to style valid or invalid form elements with zero JavaScript. If you have advanced validation requirements that are not provided natively, the new Constraints API offers a standardized JavaScript API for testing for the validity of form fields, along with a new event that you can use to detect when invalid data is entered in a field.

To demonstrate how to use the new HTML5 form features, we will work on a sample application, a registration form, which uses several of the new input types, attributes, and validation features. Before we dive into these topics in more detail, let's lay out the foundation of the sample application. The application code is contained in three files, index.html, which defines the markup, style.css, which defines the CSS styling properties, and app.js, which contains the JavaScript code for the application. All of these files should be stored in the same directory on your Web server. The basic template for the index.html file is shown in the following listing.

Listing 2.1. index.html Basic HTML structure for the registration form sample application

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Registration Form</title>
    <link rel="stylesheet" href="style.css">                          #1
    <script src="app.js"></script>                                    #2
</head>
<body>
    <h1>Awesome HTML5 Registration Form</h1>
    <form name="register">                                            #3
    <fieldset>
        <input type="submit" name="submit" value="Submit Registration">
    </fieldset>
    </form>
</body>
</html>
```

#1 App stylesheet
#2 JavaScript file
#3 Form container

The code in the previous listing uses many concepts covered in Chapter 1, such as the HTML5 doctype, the shorter form for character set <meta> elements and so on. It also loads the CSS #1 and JavaScript #2 files that we will create later in this chapter. As you work

through this chapter, you will add more markup to the file, all of which will be placed inside the <form> element #3. The final application will look like the screenshot in figure 2.1.

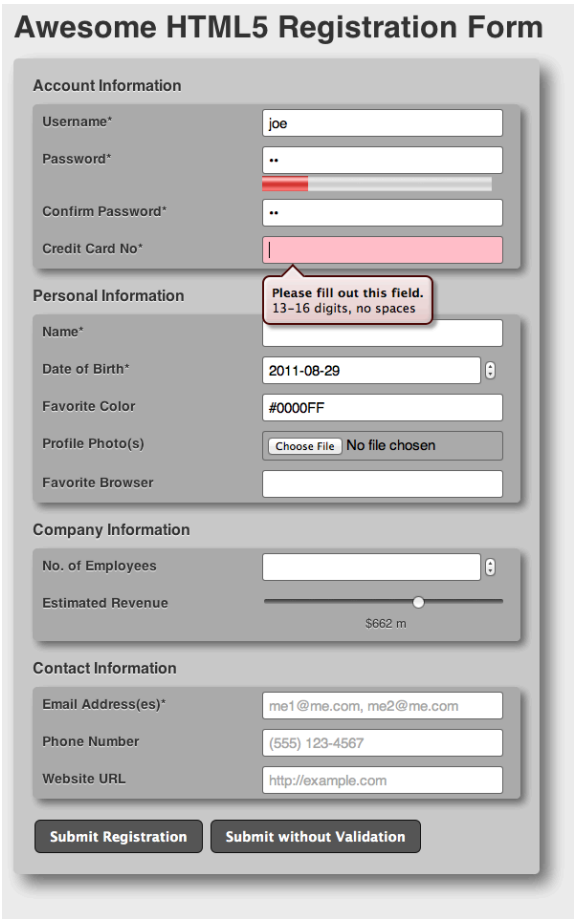

Figure 2.1. The HTML5 registration form sample application includes many new Web form features introduced in HTML5, including new input types, validation, placeholders, meters and more

> **NOTE** For the sake of brevity, we will not cover the CSS declarations required to style the form shown in figure 2.1. All of the CSS required for this application is included in this chapter's source code, available for download on manning.com.

In this chapter, you will learn about all of the new aspects of Web forms in HTML5, with fully functional examples to show you how to use them in your own applications. We kick things off with the thirteen new input types and how they can be added to existing applications

without fear that they won't work in older browsers. Next, you will discover the various new attributes that can be used on form fields to enhance them with native validation and more. Following this, you will learn about the new HTML5 elements that allow you to create features like autosuggest text boxes and progress bars, with no JavaScript code in sight. You will then discover the new set of CSS3 pseudo-classes that can be used to style your validated forms, and the Constraints API that allows you to use the power of JavaScript to enhance your form validation routines even further. Finally, you will learn how to detect whether a visitor's browser supports all of these new features, and how to provide alternative fallbacks if they don't. Let's jump right in and look at the new input types.

2.1 Changing the face of Web forms with new input types

The <input> element is used in HTML to add fields like text boxes, check boxes and radio buttons to a form. The type attribute is used to indicate to the browser what type of form control should be displayed to the user. In previous versions of HTML, there were ten different types that could be used, as outlined in Table 2.1.

Table 2.1. Valid input type values in previous versions of HTML

button	checkbox	file	hidden	image
password	radio	reset	submit	text

The problem with the types in Table 2.1 is that they don't offer any features for inputting data types like numbers, dates, colors, email addresses and URLs. Sure, you could just use a standard text input, but you would then need to use JavaScript to validate the data type on the client side. If JavaScript is turned off, this validation won't take place. To solve this problem, HTML5 defines thirteen new input types, as shown in Table 2.2.

Table 2.2. New input type values introduced in HTML5

color	date	datetime	datetime-local	email
month	number	range	search	tel
time	url	week		

In this section, you will learn about each of these new input types, how they look in browsers that have implemented them, and the benefits they provide over using the basic text input type. You will also use them in a sample Registration Form, which we will use throughout this chapter to illustrate each of the various enhancements in HTML5 web forms.

2.1.1 email, url and tel

One of the most common uses of text fields in Web forms is to ask the user to enter their e-mail address. Whether it is used in a registration form, contact form, user login, subscribing to a newsletter or posting a comment on a blog post, e-mail address input is required in the majority of forms on the Web. Of course, all e-mail addresses must adhere to a certain format in order to be deemed valid, and in most cases developers use some form of JavaScript regular expression testing to check the validity of e-mail address on the client-side. The problem with this is that different people have different opinions on what constitutes a valid e-mail address, and as a result developers use a widely varying range of different patterns in their validation routines. HTML5 eradicates this problem by passing responsibility for validating e-mail addresses to the browser itself.

> **WARNING** This chapter introduces a lot of concepts surrounding client-side data validation. You should never rely solely on client-side validation, as a malicious user can very easily bypass it. Your applications should always provide solid server-side validation at the backend, with client-side validation only used to enhance the user experience (and to reduce the load of invalid requests being sent to the server). Server-side validation is outside the scope of this book as implementations will vary between the various server side languages.

Using the email input type is extremely straightforward (as is using most of the new HTML5 input types). It works exactly the same as a regular text input, but instead of using `type="text"`, you use `type="email"` instead.

```
<input type="email" name="email_address">
```

Easy, right? When you view an email input field in a compatible browser, it doesn't actually look any different than a regular HTML text input. In fact, if you view it in any browser, it will just appear as a standard text box. One of the nice things about the new input types is that if a browser doesn't recognize them, it will just display a normal text input field instead, so you can safely add these new fields to your applications today, without worrying about them wreaking havoc in older browsers.

At this point you might be wondering, "What is the point of this new input type if it just looks like a standard text input?" The real power behind the e-mail type is that by default, a compatible browser will check that the value of the field is a valid e-mail address when the form is submitted. If the field value is invalid, form submission will be stopped and an error message will be displayed to the user, with zero JavaScript code required to do so. Figure 2.2 shows this error message in several different Web browsers.

Figure 2.2. Native e-mail address validation in Opera 11.50, Google Chrome 12 and Mozilla Firefox 5.01

In addition to the new email input type, there are two similar new types: url and tel. The url type behaves the exact same way as the email type, albeit displaying a different error message to denote that the user has entered an invalid URL. The following code is an example of how you can use the url type in HTML5.

```
<input type="url" name="web_site_address">
```

The tel input type is intended to allow users to enter telephone numbers. Unlike email and url, however, browsers don't natively validate tel input fields because there is no single valid structure for a telephone number. One website might require the user to enter a full telephone number with international dialing code and area code; others might just want the area code. On top of this, different locales may present phone numbers in different ways. In section 2.2 you will read about how you can use the pattern attribute to specify your own validation regular expressions on tel and other fields. You can use the tel input type as follows.

```
<input type="tel" name="phone_number">
```

So if the tel input type doesn't provide enhanced functionality using validation, then what benefits does it provide over a standard text input? One of the primary reasons for using this input type is that on devices with virtual keyboards such as smartphones, the browser may display a different keyboard to the user depending on the type of input field they are using. This also applies to the email and url input types. For example, when you are entering a

value in an email input type on an iOS device, the spacebar on the keyboard is made much smaller, and beside it you will find buttons for the "@" and "." symbols to make it easier to enter e-mail addresses. For url fields, the spacebar is removed altogether and replaced with buttons for ".", "/" and ".com". In the case of tel fields, the standard keyboard is completely replaced with a numeric keypad, with easy access to symbols commonly used in phone numbers like "+", "*" and "#". You can see examples of these changes in figure 2.3.

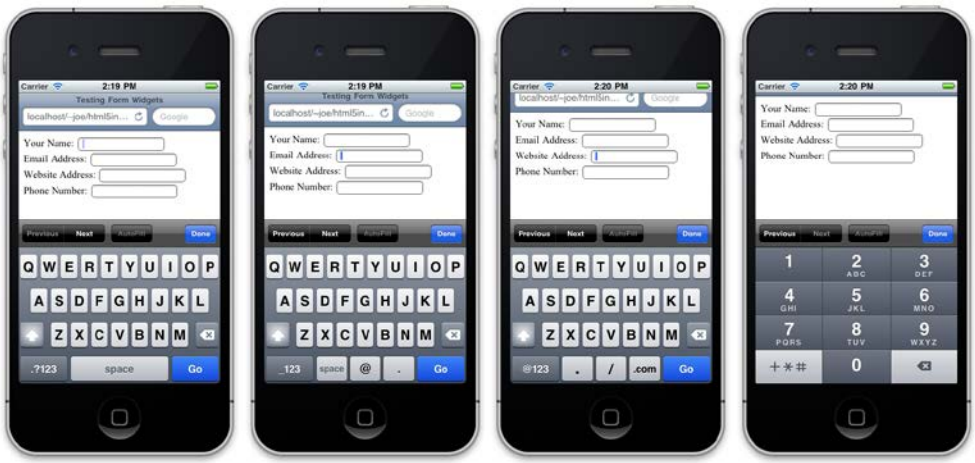

Figure 2.3. Different virtual keyboards are displayed on an iPhone for different input types – from left to right: "text", "email", "url" and "tel"

TIP You may have noticed that in the code examples for this section, we are using the <input> element without a terminating slash (like <input />). This syntax was introduced in XHTML and is no longer a requirement in HTML5. Both methods are valid, so if you prefer the XML-style convention, feel free to keep using it in your own applications.

2.1.2 number and range

In HTML 4 and XHTML, when you needed numeric data input, you were forced to use a standard text input element and validate the data using JavaScript. In addition, if you wanted to provide the user with a different user interface for a number field, such as a spinner or a slider, you had to rely on third party UI frameworks. In HTML5, there are two new input types for numeric data entry, number and range. Both of these fields have virtually the same markup and set of accompanying attributes, but they differ in terms of the UI widget that is rendered by the browser. In the case of the number field, you can expect supporting browsers to display a spinner. For example, the following code:

```
<input type="number" name="mynum">
```

produces a spinner like the one shown in figure 2.4.

> **NOTE** Mozilla Firefox (up to and including version 6.0.1, at least) does not support the number input type or any of its attributes. To add support for HTML5 form features (such as a spinner widget and numeric validation, you can use a polyfill. You will learn more about polyfills later in this chapter.

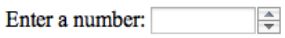

Figure 2.4. Spinner widget in Opera on Mac OS X for a number input field

The range input also stores a number value, but it displays a slider widget to the user instead. The following code:

```
<input type="range" name="mynum">
```

produces a slider as shown in figure 2.5.

Figure 2.5. Slider widget in Google Chrome on Mac OS X for a range input field

> **TIP** You may have noticed that the slider widget in figure 2.5 does not show you the value of the field. In section 2.3.1, you will learn how to use the <output> element to display this value.

If you use a number or range field as illustrated previously, you won't get an error message if you enter an invalid number in the field (although Google Chrome will actually convert non-numeric values to numeric on-the-fly as you tab through the form). However, only a valid number value will be submitted to the server by the form. For example, if you enter the value "3259a" in a number input field and submit the form, the value sent to the server will be "3259". If you enter the value "a3259", however, the value sent will be an empty string. When you think about why the browser doesn't validate by default, it makes a lot of sense, as there are so many different ways to represent a numeric value. Negative values, zero values, fractions, decimals – how does the browser determine what is valid and what is not? Instead, you need a way of telling the browser what should constitute a valid number. Fortunately, this is very straightforward in HTML5, as both the number and range input types support three new accompanying attributes that allow you to define the properties for checking that a number is valid: min, max and step.

As you might expect, the “min” attribute is used to specify the lowest acceptable value that should be allowed in the field. Similarly, the “max” attribute is used to specify the maximum acceptable value allowed. The “step” attribute allows you to set what values will be deemed acceptable between the min and max attributes. For example, if you have the following number input:

```
<input type="number" name="mynum" min="0" max="50" step="10">
```

The acceptable values will be 0, 10, 20, 30, 40 and 50. The step attribute can also be a decimal number. For example, take the following range input:

```
<input type="range" name="mynum" min="0" max="1" step="0.2">
```

In this case, the acceptable values will be 0, 0.2, 0.4, 0.6, 0.8 and 1. With the range input type, the user won’t be able to enter an invalid value as the user is presented with a slider. But with the number input type, the user can either enter the value as text or can change the value using the spin controls. If the user enters an invalid value in the text box and tries to submit the form, the browser will cancel the submit event and display an error message to the user. Figure 2.6 illustrates an example of the type of error message you can expect to see.

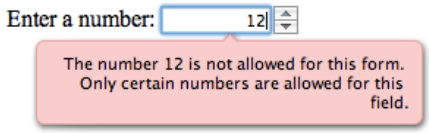

Figure 2.6. The browser (in this case Opera) will validate the number and display an error message like the one shown if it is not valid

> **WARNING** Safari 5.1 has partial support for the number input type. It will display the field with a spinner widget, but it will not validate the number against the values in the min, max and step attributes. These attributes do work correctly in the range input type, however. You will learn how to add support for the number input type for unsupported browser versions using JavaScript polyfills later in this chapter.

In addition to providing a new UI widget and validation attributes for number fields, HTML5 also defines three JavaScript APIs for interacting with number fields – stepUp, stepDown and valueAsNumber. The stepUp method allows you to increase the value of the field by incrementing it by a defined number of steps. You pass the number of steps to increment by as an argument to the method. Say for example you have a number field named mynum

with the step attribute value set to 10, and the current value of the field is 0. Executing the following code snippet:

```
document.forms.myform.mynum.stepUp(2);
```

will increase the value of the field to 20. The stepDown method works the same way, but in reverse.

> **WARNING** You must be careful when using stepUp and stepDown, as trying to step up or down outside the upper and lower bounds set by the min and max attributes will raise an INVALID_STATE_ERR exception. Another gotcha is that if you attempt to execute these methods when the field is empty, you will also raise this exception.

The valueAsNumber method allows you to get the value of the field as a floating-point number (unlike the value property of the element, which is always a string). If you use the valueAsNumber method on a field that contains non-numeric characters, it will return NaN (Not a Number).

2.1.3 date, datetime, time, datetime-local, month and week

If you have ever booked a flight or made a hotel reservation online, you've come across a date picker widget, which allows you to select the date you want to book on an easy-to-use calendar that allows you to quickly and easily jump between months and years. Nearly every website that uses date pickers seems to use a different variant, be it one that is included with a JavaScript UI library such as ExtJS or YUI, from a server-side development framework like ASP.NET, or just a standalone widget that the site's developers found on the Web or built themselves. There are a number of issues with this – the first being a lack of consistency. Some pickers will open in new browser windows as popups, others will appear over the page content, some will even show inline in the form itself (yuck!). Another problem is that almost all of these widgets (at least every one we have come across) require JavaScript in order to function. The site doesn't necessarily break without it (you can just enter a date in the textbox instead), but you won't be able to use the date picker unless your browser supports and has JavaScript switched on.

To get around these problems HTML5 defines a series of new date and time related input types that can be used to enter dates in Web forms. Unfortunately, browser vendors have been slow to add support for these new input types, and the only browser that currently provides a date picker for these elements is Opera (version 9 and later). Google Chrome has partial support for date types, oddly showing a spinner control (like used in the number input type) instead. Chrome will validate that the value is a valid date on submit, however. Apple Safari is one step back, implementing the same spinner control, but not validating the value at all.

Using the date input type is straightforward, as shown in the following snippet:

```
<input type="date" name="date_of_birth">
```

In Opera, this code will produce a drop-down box which, when opened, will display a date picker widget rather than a list of values, as shown in figure 2.7.

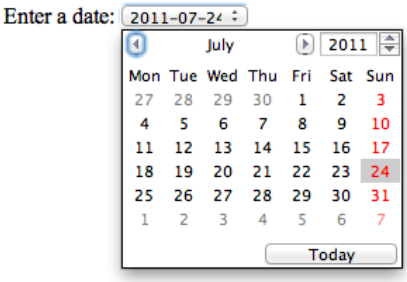

Figure 2.7. The native date picker widget in Opera 11 for Mac

In Google Chrome, that same code will produce a spinner widget, which steps up and down through dates rather than numbers. Chrome will also validate that the date entered is a valid date when the user submits the form, as shown in figure 2.8.

Figure 2.8. Date validation in Google Chrome

The format used for the date input type is YYYY-MM-DD. Other date formats are based on local conventions (for example DD/MM/YYYY is used in the UK and Ireland, while MM/DD/YYYY is used in North America). Trying to account for various locale variants would be a nightmare, so the ISO 8601 standard for Gregorian calendars is used instead. In addition to the standard date input type, HTML5 also defines a series of other date/time related types:

- **datetime:** The same as date, but also allows you to select a time (with timezone)
- **datetime-local:** The same as datetime, but without the timezone
- **time:** The same as datetime-local, but without the date segment
- **month:** The same as date, but only selects a month/year, not a day; an example of the value of a month input type is 2011-07 (July 2011)
- **week:** The same as date, but selects only a week/year, not a day; an example of the

value of a week input type is 2011-W28 (Week 28 in 2011)

In addition to the new input types, HTML5 also specifies a new API method, valueAsDate, which allows you to get the value of an input field as a Date object. This method is useful if you want to do more complex validation with your dates. For example, you might allow the user to select a start date and end date in your form. Of course, the end date should be a date on or after the start date. In the past, you would have had to get the value of the form field and construct a Date object from that value. Now, you can use the valueAsDate method to get a Date object straight from the field itself.

2.1.4 search and color

On most dynamic Web sites and applications, you will find some form of search box that allows you to find specific pages or data, typically implemented using a text input field. In HTML5, there is a new search input type which, when implemented further by browsers, should allow native enhanced functionality for your site's search, without requiring additional JavaScript development on your part. Using the search input type is similar to using any other HTML5 form field.

```
<input type="search" name="mysearch">
```

In Opera, Chrome and Safari, you will find that search inputs are styled by default to have rounded corners. In Chrome and Safari, if you enter a value in the field, a small "x" will appear on the right-hand-side of the field which, when pressed, will clear the contents of the field. An example of this can be seen in figure 2.9.

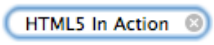

Figure 2.9. Entering a value in the search input field in Google Chrome and Apple Safari results in a small grey "x" showing, allowing you to easily clear the field's contents

The final new input type introduced in HTML5 is the color type. This field type is designed to allow you to accept a color value, in the hexadecimal (hex) notation, such as #000000 (black). The specification specifically states that the color value must:

1. Start with the # character
2. Have six characters in the range 0-9, a-f and A-F that follow the # character

WARNING You should not use color names in a color input field, as HTML5 only supports using the hexadecimal notation and the field won't work with any other values.

You can use the color input type as follows.

```
<input type="color" name="mycolor">
```

In all browsers except Opera, the color input type will appear as a standard text box, with most browsers not implementing any additional functionality. Google Chrome validates that the value entered in a color field is a valid color value according to the aforementioned rules. Opera has much more mature support for the color input type, however, and implements a native color picker widget, removing the need to implement a third-party JavaScript color picker. An example of the Opera color picker in action can be seen in figure 2.10.

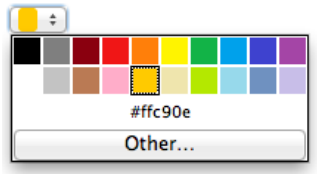

Figure 2.10. Opera color picker widget in action on Mac OS X. Clicking the "Other…" button will launch the native Mac OS X color utility, where you can choose from a much wider range of colors.

Now that you've seen the new input types available to you in HTML5, you're more than ready to get your hands dirty and apply what you've learned in a practical example. In the next section, you'll do just that, building the registration form we showed you in figure 2.1.

2.1.5 Adding form fields to the registration form sample application

As shown previously in figure 2.1, the registration form you are building in this chapter has four sections of input – Account Information, Personal Information, Company Information and Contact Information. Now that you have seen the various new types of form fields you can use in HTML5, let's start building these sections up with the appropriate types of input field. Listing 2.2 provides the code you need for the Account Information and Personal Information sections; add it to index.html, just beneath the opening <form> element you added previously.

Listing 2.2. index.html Adding the Account Information and Personal Information sections to the registration form, using new date and color input types

```
<fieldset>
    <legend>Account Information</legend>
    <ul>
        <li>
            <label for="username">Username*</label>
            <input name="username" id="username">
        </li>
        <li>
            <label for="password">Password*</label>
            <input type="password" name="password" id="password">
        </li>
        <li>
            <label for="confirm_password">Confirm Password*</label>
```

```
            <input type="password" name="confirm_password"
id="confirm_password">
        </li>
        <li>
            <label for="credit_card">Credit Card No*</label>
            <input name="credit_card" id="credit_card">
        </li>
    </ul>
</fieldset>
<fieldset>
    <legend>Personal Information</legend>
    <ul>
        <li>
            <label for="name">Name*</label>
            <input name="name" id="name">
        </li>
        <li>
            <label for="date_of_birth">Date of Birth*</label>
            <input type="date" name="date_of_birth" id="date_of_birth"> #1
        </li>
        <li>
            <label for="color">Favorite Color</label>
            <input type="color" name="color" id="color" value="#0000FF">#2
        </li>
        <li>
            <label for="photo">Profile Photo(s)</label>
            <input type="file" name="photo" id="photo">
        </li>
        <li>
            <label for="browser">Favorite Browser</label>
            <input name="browser" id="browser">
        </li>
    </ul>
</fieldset>
```

#1 date input type
#2 color input type

At this point, you are probably thinking “Where are all the new elements? This is all regular old HTML!” And you’d be right - the majority of the code in listing 2.2 contains old input types. But don’t worry – all of these old fields have a purpose in this chapter and you will be using HTML5 to improve them shortly. In the Personal Information section, you ask the user to enter their date of birth, presenting them with a new date input field #1. The form also includes a field for the user’s favorite color, using a color input type #2. Every field in the Company Information and Contact Information sections of the form are new HTML5 input types. Let’s add them now – insert the code in listing 2.3 directly after the code you added previously.

Listing 2.3. index.html Adding the Company Information and Contact Information sections to the form, using number, range, email, tel and url HTML5 input types

```
<fieldset>
    <legend>Company Information</legend>
    <ul>
        <li>
```

```
            <label for="employees">No. of Employees</label>
            <input type="number" name="employees" id="employees" min="0"
max="100000" step="100">                                                    #1
        </li>
        <li>
            <label for="revenue">Estimated Revenue</label>
            <input type="range" name="revenue" id="revenue" min="0"
max="1000" step="1" value="0">                                              #2
        </li>
    </ul>
</fieldset>
<fieldset>
    <legend>Contact Information</legend>
    <ul>
        <li>
            <label for="email">Email Address(es)*</label>
            <input type="email" name="email" id="email">                    #3
        </li>
        <li>
            <label for="phone_number">Phone Number</label>
            <input type="tel" name="phone_number" id="phone_number">        #4
        </li>
        <li>
            <label for="website">Website URL</label>
            <input type="url" name="website" id="website">                  #5
        </li>
    </ul>
</fieldset>
```

#1 number input type
#2 range input type
#3 email input type
#4 tel input type
#5 url input type

In the Company Information section there are two fields, both using new HTML5 input types. The first field is for the number of employees in the company, defined as an HTML5 number input field, with a minimum value of 0, a maximum value of 100,000 and which steps up and down in blocks of 100 #1. The second field asks the user to enter the company's estimated revenue, with a range input type allowing values of between 0 and 1,000 (million) #2. You may notice that you cannot see the current value of the field on the slider (as shown in figure 2.11)– you will resolve that issue later in the chapter when you learn about the <output> element.

Figure 2.11 The HTML5 slider widget does not display the current value of the range input field. We will see how to display this value using <output> later in this chapter.

In the final section of the form, Contact Information, you ask the user to enter their email address(es) using an email input field #1, phone number using a tel field #2 and finally the URL of their website using a url input field #3. At this point, the registration form markup will give you a result that looks similar to the screenshot you saw previously in figure 2.1. Of course, there is still quite a bit of functionality missing from the form, which you will be adding as we progress through the chapter.

In the next section, you will learn about some of the new form attributes introduced in HTML5, including ones that enable you to implement JavaScript-free client-side validation features such as mandatory fields and regular expression pattern matching.

2.2 Kicking the JavaScript habit with new attributes

In addition to the new input types covered in the previous section, HTML5 defines numerous new attributes that you can add to your form fields to provide additional functionality to your users, which would previously have required JavaScript code, and others that were not possible at all (for example attaching multiple files to a single file input field) without some third-party browser plugin such as Adobe Flash or Java applets.

In this section, you will learn about these new attributes, what functionality they provide, and how you can use them in your own pages and applications. You'll learn about how multiple and accept improve the file input type, allowing us to accept multiple field uploads without providing the user with multiple upload fields. Then, you'll learn how to build native JavaScript-free form validation using the required and pattern attributes. After that, you'll discover how to by-pass this validation using the novalidate and formnovalidate attributes. As we wrap up, we'll investigate several new attributes that allow you to modify a form's behavior from <input> elements, rather than having to change the <form> element itself. But first, let' s look at placeholder and autofocus, which provide two features common to modern Web forms but that, up until now, you could only implement using JavaScript.

2.2.1 placeholder and autofocus

Suppose you are building a Web application that provides a search feature to allow users to easily find something. You want to place the search box in a prominent position on the Web page, but it can't take up too much space. The first thing to go is the submit button – most users will press enter after entering a search term, and since "enter" submits the form, no button is needed. But what about that pesky "Search this site" label that tells the user what this field is? Wouldn't it be great if you could get rid of this label altogether and instead put descriptive text in the field itself, only for it to be replaced as soon as the field receives focus (becomes the active field)?

In previous versions of HTML, you would have implemented the space-saving feature using a combination of CSS styling and JavaScript code, populating the value with a standard placeholder and setting the text to a lighter color when it is not focused and is empty. When the field received focus, you would then dynamically darken the text color and clear the field's value. When the field lost focus again, you would check if it was empty, and if so, you

would re-style it so the text is once again lighter, and you would set the value back to the placeholder text. Because it uses the value field for the placeholder, you would then need to build the placeholder text into your client-side validation routines, assuming the field is empty if its value matches the placeholder text. Talk about a lot of hassle for such a simple feature!

Thankfully, in HTML5, the placeholder attribute allows you to provide the same functionality by defining a string of text that will appear in an input field when it does not have focus and its value is empty.

```
<input name="first_name" placeholder="Enter your first name">
```

Support for the placeholder attribute is present in the latest version of every major Web browser, so there's no reason why you can't use it right now. An example of this attribute in action can be seen in figure 2.12.

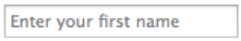

Figure 2.12. The placeholder attribute in action in Firefox 5.01 on Mac OS X

When you give the field focus, the placeholder text will disappear and the field will be empty. If you then enter text in the field and move focus away from the field, the value you entered will persist. If you leave the field blank and move focus away, the placeholder will be restored. Similarly, if you clear the value of a field with a placeholder defined and then move focus away from that field, the placeholder will display.

When building Web pages that primarily consist of a form, it makes sense that focus should be given to the first field in the form. Up until now, to facilitate this, you would need to use the focus() JavaScript method to set the focus to the desired element after the document had loaded. In HTML5, you can simply use the autofocus attribute instead, as demonstrated in the following code.

```
<input name="first_name" autofocus>
```

The autofocus attribute has a Boolean value, and as a result you can use it without explicitly specifying a value. If you prefer to use XML syntax, the correct way to use it is autofocus="autofocus".

2.2.2 multiple and accept

Web applications can offer users the ability to upload files to the Web server using the file input type in HTML. This produces a form field that allows the user to select a file from their filesystem, which will be uploaded to the server when they submit the form. This works great if you want to upload a single file, but if you want the user to be able to upload multiple files,

previously you would have needed to display multiple file input fields – one for each file the user wanted to upload. Not only is this visually unappealing, but it also means that if you do not know how many files the user wishes to upload, you need to use some form of scripting to dynamically produce new fields on the fly. Many applications (including the popular blogging engine, Wordpress) turned to Adobe Flash for a solution to this problem, embedding a Flash movie in the page for "advanced upload", meanwhile providing a fallback link to a "basic upload" option, which used regular HTML file input types.

Thankfully, HTML5 allows us to accept multiple files using a single form field via the "multiple" attribute. You may be familiar with the "multiple" attribute on the <select> element, which allows you to select multiple items from a list box. In HTML5, this attribute is now also available on the <input> element for file input types. For example, if you are allowing users to upload photos on a form, and you want to allow them to upload more than one photo, you can use syntax such as the following to do so:

```
<input type="file" name="photos" multiple>
```

When you first load this code in your Web browser, you probably won't notice any difference from a regular file input field. However, when the dialog window that allows the user to select files opens, he/she can now select multiple files rather than just one.

> **TIP** On Windows, you can select multiple items by holding Ctrl and clicking each item you wish to select. Mac users can do this by holding Cmd and clicking each item. Alternatively, to select a large group of adjacent files, click the first item in the group, then hold Shift and click the last item in the group, and all items between will also be selected. This should work in both Windows and Mac OS.

In Chrome and Safari, when the user selects multiple files, the label next to the "Choose file" button will display the number of files selected. Safari adds a nice extra touch by displaying a file stack icon to denote multiple files have been selected. In Firefox, the text box that displays the filename contains the full path for each selected file, separated by a comma. Opera behaves in a similar manner, but instead separates each file path with a semi-colon, wrapping each path in double quotes. Unfortunately, as of version 9 of Internet Explorer, the "multiple" attribute for file input types is not supported. Figure 2.13 illustrates how multiple files are handled in each of the aforementioned browsers.

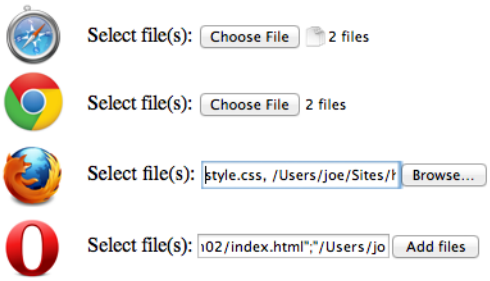

Figure 2.13. Accepting multiple files using a single input field in Safari 5.1, Chrome 12, Firefox 5.01 and Opera 11.50

The "multiple" attribute is also available on the new email input type, allowing users to enter multiple e-mail addresses into a single field. This is useful if you are designing a form that allows users to send e-mail to several recipients at once, as in the following example:

```
<input type="email" name="recipients" multiple>
```

In supporting browsers, you can enter multiple e-mail addresses, separated by commas, into this field. If you try to separate the addresses with any other character such as a space or semi-colon, you will get an error message. Each individual e-mail address must also be in the valid email address format (for example me@example.com), or you will trigger an error. Unfortunately, this feature is not yet supported in Safari or Internet Explorer (neither of which even properly support the email type), but it is present in current versions of Chrome, Opera and Firefox. figure 2.14 shows an example of the error message you receive if you do not separate addresses with a comma in Google Chrome.

Figure 2.14. Client-side validation of multiple addresses in an email input field

In the past, if you wanted to restrict the types of files users could upload to the Web server, you would need to do so on the server-side, after the file has been uploaded. Sure, you could use JavaScript to check the filename extension, but that only works after a file has been selected. Besides, malicious users can bypass this fairly easily by renaming files to have a different extension. In HTML5, you can define the MIME types that can be selected, to stop

users from selecting unsupported file types in the first place. This functionality is provided via the new “accept” attribute, the value of which can be set to a comma-separated list of MIME types.

> **WARNING** The accept attribute is useful for ensuring users don’t mistakenly select the wrong type of file, but you should NEVER trust it as your only source of validation, as it can be very easily bypassed. You should always check the type of the file on the server-side, as this is the only way to be absolutely positive that a file is of a particular type.

The following is an example of how you would use the “accept” attribute to specify that users may only select JPEG, GIF and PNG image files when using a file input field.

```
<input type="file" name="photo" accept="image/jpeg,image/gif,image/png">
```

When the user opens the file dialog in a supported browser, they will only be able to selected supported file types. On Mac OS, files of the wrong type will be greyed out so the user can’t select them in the file dialog, as illustrated in figure 2.15. On Windows, any files of the wrong type will not appear at all in the dialog.

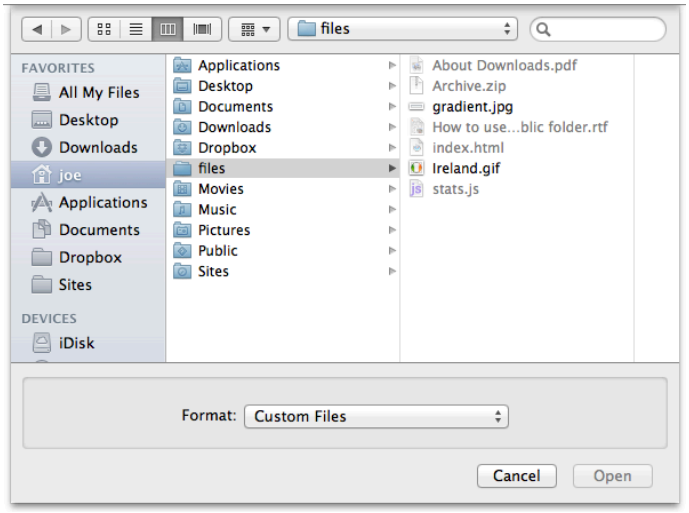

Figure 2.15. Preventing users from selecting unsupported file types using the accept attribute. In the figure, you will notice that non-image files are greyed out and cannot be selected.

Mac OS X 10.7 Lion users can select the “All My Files” option in the file selection dialog, and the results will be pre-filtered to only include the types of files specified in the “accept” attribute. Pretty neat, huh?

2.2.3 required and pattern

In the previous sections, we saw how some of the new input types (such as email and color) add standard validation performed natively by the Web browser. This is very useful for performing basic client-side sanitization without relying on JavaScript. In fact, HTML5 enables native validation to be performed on all types of form field using two new attributes – "required" and "pattern".

The "required" attribute is self-explanatory – the user must enter a value for the field in order to submit the form. If the user leaves the field blank, the browser will display an error message for that field and halt the form submission. The following snippet shows how you would make a username text field mandatory using the required attribute:

```
<input name="username" required>
```

Support for the required attribute was introduced in Internet Explorer 10, Firefox 4, Chrome 10, and Opera 10.6. Safari has had partial support for the attribute since version 5.0.

Checking if a field is empty is a basic form of validation, but in many cases you'll want to perform more complex data checks to ensure that the user has entered data in the correct format. For example, if a user were entering a credit card number, you would want to check that this field only contains numbers and is between 13 and 16 characters in length. In the past, to perform this check on the client-side, you would need to use JavaScript. In HTML5, however, you can implement regular expression testing on form fields natively using the pattern attribute. Take our example of a credit card number field, which should only accept digits and should be between 13 and 16 characters long. You could specify this validation test using the following markup:

```
<input name="cc_number" required pattern="[0-9]{13,16}" title="Numbers
only, no spaces, 13-16 digits long">
```

If the user enters an invalid value, supporting browsers will display an error message and prevent the form from submitting. You can specify a message using the "title" attribute, which will be displayed in the error message to indicate to the user why their entry was not accepted. Figure 2.16 illustrates how this appears in Mozilla Firefox 5.01.

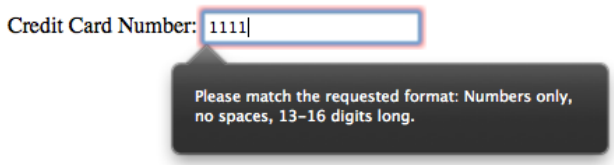

Figure 2.16. Restricting user input using regular expressions and the pattern attribute

The pattern used in this example is very basic – if you need more complex validation you can define highly specific regular expressions to test your fields against. For a wide range of examples, see http://html5pattern.com/ or visit http://www.regular-expressions.info/ for an in-depth guide to regular expressions in general.

2.2.4 novalidate and formnovalidate

Suppose you are developing a complex JavaScript application and want to prevent the browser from performing native validation of your form fields. Maybe you want to use your own validation routines to provide error messages consistent with your application's look and feel. You can implement this functionality in HTML5 using the novalidate attribute on the <form> element.

To use the novalidate attribute, simply add it to your <form> element as follows:

```
<form name="myform" novalidate>
```

With this flag set in your markup, browsers that support native form validation will not perform any validation for this form. This means that any required fields, fields with pattern attributes, or input types that have a specific data type (number, email, etc.) will not be validated by the browser itself.

Another scenario for which you may want to prevent form validation is when you need to allow the user to save their work for later completion, for example, when they're filling out a form that collects large amounts of data, such as an online loan or graduate school application, or when they're composing an email message and saving it as a draft. In the latter case, if the user saves the message, it is likely because they need to exit your email application quickly; the application shouldn't validate that they have entered a recipient and subject, as the user might not have time to correct their errors. If and when the user sends the message, then you should validate the fields. This functionality is possible in HTML5 using the formnovalidate attribute on your form submission <input> elements.

To integrate such functionality in an email application, add the formnovalidate attribute to the button element for which you wish to ignore validation when pressed. For example, the following code illustrates two submit input elements, one that submits the form, the other that saves it, bypassing the browser's validation routines.

```
<input type="submit" name="submit" value="Send Message">
<input type="submit" name="submit" value="Save as Draft" formnovalidate>
```

If the user clicks on the "Send Message" button, and invalid form fields exist, in this case a missing "to", "from", or perhaps even a missing "subject", the browser will stop the form from submitting and will display an error message. If the user clicks on the "Save as Draft"

button, the browser will ignore any invalid fields and submit the form regardless. Then, on n the server-side, you check the value of the “submit” field to determine whether the data should be stored as a submitted or saved item (after you have performed additional server-side validation, of course!).

2.2.5 form, formaction, formenctype, formmethod and formtarget

In order for a form element’s value to be sent to the server when the form is submitted, the element must be a part of the form, typically by being nested inside the <form> element itself. There may be cases, however, where you would like to have a form field that lives outside of the main <form> block. Imagine, for example, that you are building an online shopping cart application where you have multiple forms on the page – one for modifying your cart (emptying the items, calculating the shipping cost, increasing quantities, and so on) and another for checkout with a payment processor. You may want to group the buttons for both forms together, but in HTML 4 you would not have been able to do so without some messy CSS positioning, as the buttons would have to be placed within a <form> block.

In HTML5, however, the new “form” attribute allows you to indicate that a form field belongs to a particular form. For example, if you have a form with an ID “myform”, you could create a field that exists outside the context of that form but still gets sent to the server along with it, with the following code:

```
<form id="myform">
    <!-- Main form elements go here -->
</form>
<input name="another_field" form="myform">
```

Earlier in this section, you learned how the formnovalidate attribute allows you to tell the form not to validate when a particular button is pressed. HTML5 also defines a series of attributes that allow you to change various properties of the form based on the button the user presses:

- formaction
- formenctype
- formmethod
- formtarget

These attributes allow you to modify the value of the action, enctype, method and target attributes, respectively, on the form the button belongs to. This is demonstrated in the following code for a login form that allows a user to sign up if not already registered:

```
<form method="post" action="/login">
    <label>Email Address</label> <input type="email" name="email"> <br>
    <label>Password</label> <input type="password" name="password"> <br>
    <input type="submit" value="Login">
```

```
    <input type="submit" value="Sign Up!" formaction="/signup">
</form>
```

With this code, if the user clicks the “Login” button, the default form attributes are used; if the user presses the “Sign Up!” button, the form sends form data to a different URL. Previously, you would have needed two separate forms to provide this functionality, or alternatively you could have dynamically changed the form’s attributes using JavaScript. In HTML5, however, you can now implement this functionality natively, with zero dependence on JavaScript. Now your web forms can be leaner and smarter. Your users save time. And you ease the load on your Web server in the process. Good stuff!

Note that these form attributes work in the Firefox 4 and later, Chrome 10 and later and Opera 9.5 and later, but will not work in Safari (up to and including version 5.1) and Internet Explorer (up to and including version 9).

2.2.6 Improving the registration form with HTML5’s new form attributes

When we last left it, your registration form application had all the elements needed to allow users to input data. You can improve on that basic design greatly by adding the enhanced functionality of the new form attributes introduced in HTML5. Let’s start off by adding the required attribute to all of the mandatory fields in the form. Take the username field, which is currently defined using the following HTML markup:

```
<input name="username" id="username">
```

Let’s make this field a required field, and while you are at it, tell the page to automatically focus on this field when it loads. To make these changes, use the following code:

```
<input name="username" id="username" required autofocus>
```

Add the required attribute (but not the autofocus attribute) to each of the following fields in the form:

- Password
- Confirm Password
- Credit Card No
- Name
- Date of Birth
- Email Address(es)

While we are on the subject of validation, let’s use the new pattern attribute to apply a regular expression check to the Credit Card No. field. This field is currently declared as follows:

```
<input name="credit_card" id="credit_card" required>
```

To specify that this field should only allow numbers and must be between 13 and 16 characters in length, and to define an error message to be displayed in the event of an invalid value using the title attribute, expand the code as follows:

```
<input name="credit_card" id="credit_card" pattern="[0-9]{13,16}"
title="13-16 digits, no spaces" required>
```

The final validation change you will make is to define a second submit button that allows the user to bypass validation when submitting the form. In the real world, you wouldn't provide such a button on a registration form, but let's include it to demonstrate how the formnovalidate attribute works in any case. Add the following markup directly beneath the current submit button code:

```
<input type="submit" name="submit" value="Submit without Validation"
formnovalidate>
```

Next, let's change the Profile Photo(s) and Email Address(es) fields to allow the user to add more than a single item using the multiple attribute. Like the required attribute, adding the multiple attribute is straightforward. The code for the Profile Photo(s) file input field is currently as follows:

```
<input type="file" name="photo" id="photo">
```

To enable multiple files to be selected, change it to:

```
<input type="file" name="photo" id="photo" multiple>
```

Apply the same change to the Email Address(es) field, which will allow the user to enter multiple email addresses by separating each with a comma. It may not be obvious to the user that they can do this, so let's use the placeholder attribute to give the user a hint that they can enter more than one email address in this field. The code for this field, before our change, looks like this:

```
<input type="email" name="email" id="email" required multiple>
```

Change it to the following:

```
<input type="email" name="email" id="email" placeholder="me1@me.com,
me2@me.com" required multiple>
```

Other form fields that can cause confusion in terms of their format include phone numbers and URLs, so let's add a placeholder to the Phone Number and Website URL fields as well:

```
<input type="tel" name="phone_number" id="phone_number" placeholder="(555)
123-4567">
<input type="url" name="website" id="website"
placeholder="http://example.com">
```

If you load the form in an HTML5-compatible Web browser, you will see that the form has been enhanced considerably by these new attributes. In figure 2.17, you can how the placeholder attribute has added descriptive text to the input elements themselves.

Figure 2.17. The placeholder attribute has added descriptive text to the email, tel and url fields to aid the user in filling out these fields using the correct format.

In the next section, you will learn about new HTML elements that you can use to improve your forms even further, with features like native autocomplete, progress bars and meters. We'll also help you enhance your registration form using those new features.

2.3 Giving users additional views and feedback on form data

Allowing your users to enter data in forms is only part of the task at hand. In modern Web applications, you typically need to give your users instant feedback with easy-to-understand views of the data they have entered. For example, if your application has a policy for minimum password strength levels, you may want to indicate to the user, as they type, if the password they have entered meets this standard, rather than waiting for them to complete the entire form and submit it, and then returning them to the page and displaying an error message. Users don't like having to enter data unnecessarily, and any aids you can give them to make their lives easier will improve your forms considerably.

In this section, you will discover new HTML elements that allow you to give users more information about the form data they are working with. Firstly, by modifying the registration form application, you will learn how the <output> element allows you to present the value of a form field to the user, and how you can use it to show the actual current value of a range input slider control. Next, you'll see how to implement a native autocomplete text box using the <datalist> element and the list attribute. Finally, you'll see how to use the <meter> and <progress> elements to give users a graphical representation of a measurement, using the example of a password strength calculator.

2.3.1 Showing display values on the registration form with <output>

One of the problems with using the new range input type in HTML5 is that the slider control it produces does not give the user any indication of the actual value of the field. Wouldn't it be better, if instead of merely outputting the raw value, you could use a separate element to display the value in a manner that you specify? We think so too, and the HTML5 specification dictates that you use an <output> element to do so. We're going to show you how <output> works by modifying your registration form application. The registration form has a range slider that allows the user to select the revenue their company earned last year. Let's add an <output> element to display its value. In the index.html file, find the following line:

```
<input type="range" name="revenue" id="revenue" min="0" max="1000" step="1"
value="0">
```

Directly beneath it, add the following markup:

```
<output for="revenue" name="revenue_display">$0 m</output>
```

This code defines an <output> element that displays information about the element defined with the ID specified in the for attribute, in this case "revenue". It would be nice if this was all you needed to do to connect your input field to the <output> element but, unfortunately, you need to perform one more step to join those elements. You need to listen for changes to the revenue slider, and update the revenue_display element with a new value every time the slider value is changed. You can do this using the new "oninput" event, which is fired when an input field's value changes, as shown in the following code snippet. Add this code to a new file named app.js and store it in the same directory as the index.html file you've been working with throughout this chapter.

```
window.onload = function() {
    document.register.revenue.oninput = function() {
        var revenue = parseInt(document.register.revenue.value, 10);
        if(revenue < 1000)
            document.register.revenue_display.value = "$"+revenue+" m";
        else
            document.register.revenue_display.value = "$"+(revenue/1000)+"
bn";
    }
}
```

Now, when the user slides the range input field to change the value, they will see the value in the output element change in tandem, nicely formatted to show the currency symbol and relevant suffix (m for million or bn for billion) depending on the value of the field. An example of the slider in action is illustrated in figure 2.18.

Figure 2.18. Displaying the value of a range slider using the <output> element

The <output> element's usefulness is not limited to merely showing the value of a range slider. You can also use it to display the result of any calculation. For example, if you had two number input fields with the IDs field_a and field_b, you could display the sum of those two values in an <output> element. You can indicate the related <input> elements by specifying the ID of each related element, separated by spaces, in the "for" attribute of the <output> element.

2.3.2 Adding native autocomplete to the form with <datalist> and the list attribute

Autocomplete fields are common in modern Web applications. Think of a Google Suggest-style search box, where suggestions for search terms are displayed as you type your query. Implementing this type of functionality was previously quite difficult, and involved using a combination of HTML, CSS and JavaScript to dynamically display on the page an element that would give the user a list of options to select from as they typed into a text box. Things got even trickier if you wanted to ensure the user could use their keyboard to navigate through the available options. In short, providing this somewhat basic feature required some very complex code.

In HTML5, there is a new element, <datalist>, that allows you to easily create lists of items, which can then be designated as the list of potential autocomplete items for a given form field. The <datalist> element typically consists of one or more <option> elements, just like you'd find in a <select> element. The difference is that a <datalist> element does not actually render as a form widget, but is rather "attachable" to a regular input element by means of the list attribute.

Since the best way to see how <datalist> works is by example, let's use the element to create an autocomplete field on your registration form. Find the following line in index.html:

```
<input name="browser" id="browser">
```

Replace that line with the following lines of code:

```
<input list="browsers" name="browser" id="browser">
<datalist id="browsers">
    <option value="Chrome"></option>
    <option value="Firefox"></option>
    <option value="Internet Explorer"></option>
    <option value="Opera"></option>
    <option value="Safari"></option>
</datalist>
```

In this example, you define a standard text input field with the name “browser”, where the user can enter the name of their favorite Web browser. This field has a list attribute with the value “browsers”. This value corresponds to the <datalist> element that follows, which has the ID “browsers”, and contains five of the most popular Web browsers on the market.

If you load the example in a supported Web browser and begin to type the name of a Web browser, you should see the list of suggested options appear, as shown in figure 2.19.

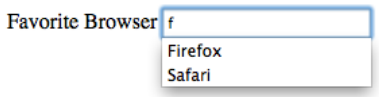

Figure 2.19. Native autocomplete in Firefox 5.01 with <datalist> and the list attribute

Browsers have been quite slow to implement <datalist> and the list attribute, with Firefox 4+ and Opera 10.6+ being the first to support it. Chrome as of version 12 has no support for this feature, and though Safari recognizes the list attribute, it does not do anything with it.

> **TIP** Applications should only use an autocomplete feature when appropriate, and not as a standard replacement for <select> drop-down boxes. Autocomplete is great when the user can enter any value but you want to provide helpful suggestions. Standard drop-down boxes are better if the user must select a value from a pre-defined list of options, and cannot enter a value of their own.

Oddly, the implementation of this feature varies between Firefox and Opera. In Firefox, if you type “f”, you will see matches for “Firefox” and “Safari” because both contain the character “f” somewhere in the text. In Opera, however, you will only see “Firefox” as it begins with the character “f”. If this inconsistency will cause issues in your application, you might be better off using a fallback method that works in the same way in all browsers instead. You will learn more about fallback techniques later in this chapter.

2.3.3 Presenting values graphically with <meter> and <progress>

Many forms of data can be presented in a form of scale using a measuring instrument or meter. For example, to visually represent temperature, you would use a thermometer. To represent speed, you use a speedometer. To represent mass, you might use weighing scales. In HTML5, you can use the <meter> element to represent a measurement within a given range or a fractional value. For example, you might use <meter> to show a star rating for a product, with zero stars being the lowest value and five stars being the highest. You could also use <meter> to indicate how much disk space a user has in a storage system, or how many days are left before an account expires.

The <meter> element supports various attributes:

- **value**
  The current value of the measurement
- **min**
  The minimum possible value
- **max**
  The maximum possible value
- **low**
  The low part of the range (must be less than or equal to the high value)
- **high**
  The high part of the range
- **optimum**
  The optimal desired value between min and max

Let's see how the <meter> element works by using it to illustrate the strength of a user's password in your registration form. The scale for this measurement will be from zero to five, with each value having the following meaning and properties:

- **No value entered (0)**

  User has not entered any value
- **Very Weak (1)**

  Password is between 1 and 3 characters long
- **Weak (2)**

  Password is between 4 and 5 characters long
- **Medium (3)**

  Password is between 6 and 7 characters long
- **Strong (4)**

  Password is between 6 and 7 characters long and contains at least one number
- **Very strong (5)**

  Password is 8 or more characters long and contains at least one number

Now, add the <meter> element directly beneath the password field in the registration form. Find the following line in index.html:

```
<input type="password" name="password" id="password" required>
```

Add the following code directly beneath this line.

```
<meter id="password_strength" value="0" min="0" max="5" low="3" high="4"
optimum="5"></meter>
```

This code adds a required password field to the form, and a password_strength meter element, which has a range of zero to five, with an initial value of zero. A value of 3 is considered low, 4 is considered high and 5 is considered optimum.

In order to calculate the strength of the password, you need to write a JavaScript function that will check the value of the field and determine its strength based on the properties in the previous list. This function needs to be fired when the password is input, so attach it to the "oninput" event using the following code, which should be added to the app.js file, inside the window.onload function you created previously, just before the closing brace.

```
document.getElementById("password").oninput = function() {
    var password = document.register.password.value, strength = 0, len =
password.length;
    if(len > 0) strength = 1;
    if(len > 3) strength = 2;
    if(len > 5) strength = 3;
    if(len > 5 && /\d/.test(password)) strength = 4;
    if(len > 7 && /\d/.test(password)) strength = 5;
    document.getElementById("password_strength").value = strength;
}
```

NOTE The algorithm used in this example for checking the strength of passwords is not a very scientific one, and is meant for illustrative purposes only. If you are writing your own password strength meter, you should build an algorithm that fits in with your application's password security policy.

In this example, when the user enters a password, the meter will dynamically change based on the value of the password entered. In figure 2.20, we show the result of having entered a very strong password, as indicated by the full meter widget.

Figure 2.20. Password strength checking meter in action. In this example, the user has entered a long, strong password, and the <meter> element reflects this strength by showing a full green bar.

Another UI widget commonly used in modern Web applications is the progress bar. An application will typically display a progress bar to the user when an intensive and time-consuming process is ongoing and the user needs to wait for the operation to complete. An example of this may be when uploading a file to the server or connecting to a chat operator in a helpdesk application. In some cases (such as uploading a file), there is a defined range, which allows you to compute a completion percentage value, which can then be updated to show the actual "progress" being made. In other cases (such as waiting for a helpdesk

operator), the percentage complete is indeterminate; you cannot tell the user how far through the process they are.

In HTML5, you can provide both determinate and indeterminate progress bars using the <progress> element. To use a determinate progress bar, you must supply a current value. The progress bar will be rendered at this value until such a time as the value is updated (typically using JavaScript). An example of this might be:

```
<progress id="upload_progress" value="0"></progress>
```

By default, the value of the <progress> element should be between 0 (zero) and 1 (one). You can change this by setting the max attribute. To set the <progress> element so that the range is from 0 to 100, you would set the max attribute to 100, as follows:

```
<progress id="upload_progress" value="0" max="100"></progress>
```

To update the value of the progress bar, you use JavaScript to change the value property. For example, to increase the value of the previous progress bar by 10%, you would use the following JavaScript code:

```
var prgBar = document.getElementById("upload_progress");
prgBar.value += 10;
```

Figure 2.21 shows an example of a 10% complete progress bar in Google Chrome version 12.

Figure 2.21. Progress bar widget in Google Chrome, with 10% complete

To create an indeterminate progress bar, simply use the <progress> element with no attributes at all. In Google Chrome, this will create a “never-ending” progress bar that loops over and over. An example of this progress bar can be created with the following code:

```
<progress id="operator_progress"></progress>
```

This produces a progress bar that looks like figure 2.22.

Figure 2.22. Indeterminate progress bar widget in Google Chrome

Browser adoption has been slow for both the <meter> and <progress> elements, with Firefox 6 only providing support for <progress> and not <meter> and IE 10 and Safari 5.1 not supporting either element at all. In Opera 11.50, only determinate progress bars (where there is a known percentage) work correctly, indeterminate progress bars will always display as empty.

In the next section, you'll learn about some new related pseudo-classes in CSS3 that can help you style your HTML5-enabled web forms, as well as how to use the Constraints API to provide advanced validation and client-side data testing using JavaScript.

2.4 Enhancing forms with CSS3 pseudo-classes and the Constraints API

Earlier in this chapter, we discovered several new input types and validation attributes that you can use to define rules for your form fields in HTML5. For example, if you have an email input field and the user attempts to enter an invalid email address, they will be stopped in their tracks and shown an error message to inform them of where they went wrong. In many cases, you may prefer to implement your own layer of validation on top of the native validation routines, should the user's browser support CSS3 and JavaScript. You may want to style your form fields and error messages to suit your own application, test for validity using JavaScript so you can perform form requests using Ajax rather than reloading the entire page, or perform a certain action as soon as a form field's value becomes invalid, rather than waiting for the entire form to be submitted.

In this section, you will learn how to enhance your forms with CSS3 pseudo-classes that allow you to style your form fields based on their given state at any particular time. You will then learn how to check form validity using JavaScript APIs, allowing you finer grained control over what happens when the user enters incorrect data in a form. Next, you will learn about the invalid event and how to handle it using JavaScript code. Then, we'll wrap up by adding custom styles for invalid fields in your registration form. Let's get started!

2.4.1 Styling required, valid and invalid input elements with CSS3

A pseudo-class in CSS allows you to style an element based on its state at any particular moment. A popular pseudo-class is :hover, which is usually applied to <a> elements to provide a different style when a user moves the mouse over a link. In CSS3, there are a series of new pseudo-selectors that represent the state of form fields:

- valid
- invalid
- required
- optional
- in-range
- out-of-range

- read-only
- read-write

A very simple example might be setting the border of any invalid input elements to red. To do this, you would use the following CSS definition:

```
input:invalid {
    border-color: red;
}
```

Pseudo-classes are stackable and can be used in tandem with one another. If you wanted to only style invalid fields when they have focus, you could use the following code instead:

```
input:focus:invalid {
    border-color: red;
}
```

Using these pseudo-classes, you can define styles that will be applied to form fields as the user fills out the form. Changing the appearance of form fields in this way does not require JavaScript, as the browser will change the state of the field natively. One of the problems with this approach, however, is that the form field will be styled as valid or invalid from the very start, even before the user has input anything, which does not provide for a good user experience. As a result, it may be more prudent to use JavaScript events to add CSS classes to define styles on the fly instead.

2.4.2 Checking the validity of form fields with JavaScript

In addition to providing new native validation functionality, HTML5 also defines a new constraint validation API that you can use in your JavaScript code. This API allows you to work with HTML5's native validation features, checking for the validity of fields, performing custom validation routines and changing validity if required. Let's investigate the various methods and properties the constraint validation API has to offer. The methods and properties defined here apply to each individual form element, and not the form as a whole.

- **checkValidity()**
  This method checks if an element is valid or not. If it is valid, it returns true, otherwise it returns false. If the element is invalid, it will also fire the invalid event.
- **setCustomValidity()**
  This method sets a custom error message that will be displayed to the user when the validity of the form is reported to the user. If this is set to something, the field is deemed to be invalid. The field can be set back to the valid state by setting this to an empty string.
- **willValidate**
  This property is true if the element is to be validated when the form is submitted, otherwise it is false. It does not check whether the element is valid or not, but rather

if the element's value will be checked or not when the form is submitted.

- **validationMessage**
  This property contains the error message that will be shown to the user when the element is checked for validity and fails.
- **validity**
  This property is a ValidityState object, which itself contains the following properties:
    - valueMissing
    - typeMismatch
    - patternMismatch
    - tooLong
    - rangeUnderflow
    - rangeOverflow
    - stepMismatch
    - customError
    - valid

  Each of these properties has a true or false value, depending on whether the element has passed or failed the relevant validity test. If the valid property is true, then all of the other properties must be false. Similarly, if the valid property is false, at least one of the other properties must be true.

You can use these methods and properties to do custom error handling in JavaScript, while still using the standardized API for form validation in HTML5.

2.4.3 Handling the invalid event

In addition to the new API methods and properties you discovered in the previous section, HTML5 also defines a new event, "invalid", which can have an attached listener function. This event will be fired by an element when it fails a test for validity. This can occur when a user submits a form, or when a JavaScript statement checks the validity of the element using the checkValidity() API method.

An example of how you might implement a listener for the "invalid" event is as follows:

```
document.myform.first_name.oninvalid = function(e) {
    e.preventDefault();
    alert("The first name field is invalid!");
}
```

This code will prevent the default action from running when the invalid event fires. If your Web browser outputs a standard error message, this will not display; an alert dialog message displays instead. In a real-world application, you would likely use this code to show a custom error message that fits in with your application's style. The great thing about

writing your validation in this manner is that, even if the user is using a non-JavaScript browser or has JavaScript support switched off, they will still experience some client-side validation before their form data is sent server-side. Not only does this save the user time modifying the form after the server responds with any errors, but it also saves precious bandwidth being taken up by erroneous form submissions.

2.4.4 Adding custom styles for invalid fields in the registration form

Now, with new knowledge under your hat, let's add some CSS3 styles to your application that will style the currently focused field to indicate to the user whether it is valid or invalid. If it is valid, you will give the field a light green background; if it is invalid, you will give it a pink background to make clear that it is erroneous. Add the following CSS declarations a file named style.css, stored in the same folder as index.html and app.js.

Listing 2.4. style.css Adding CSS3 validation pseudo-classes to the registration form application

```
input:not([type=submit]):not([type=range]):focus:valid {
    background: lightgreen;
}

input:not([type=submit]):not([type=range]):focus:invalid {
    background: pink;
}
```

Finally, let's add some basic JavaScript code to tell the user whether the form is valid or invalid when they submit it. You can do this using the Constraints API function checkValidity. Add the following code to app.js, inside the previously created window.onload function, just before the closing brace.

```
document.register.onsubmit = function() {
    if(document.register.checkValidity()) alert("Valid!");
    else alert("Invalid!");
}
```

When the user submits the registration form, a "Valid!" message will be displayed if the form has passed all of the HTML5 validation features you have added to it. Otherwise, an "Invalid!" message will be displayed.

Although we still need to show you how to provide fallbacks for old browsers, your registration form application is now complete. You started with a relatively standard HTML form, added some of the new HTML5 input types, attributes and presentation elements and styling, and now should have a usable, lean, smart form that looks something like the screenshot in figure 2.23. Congrats!

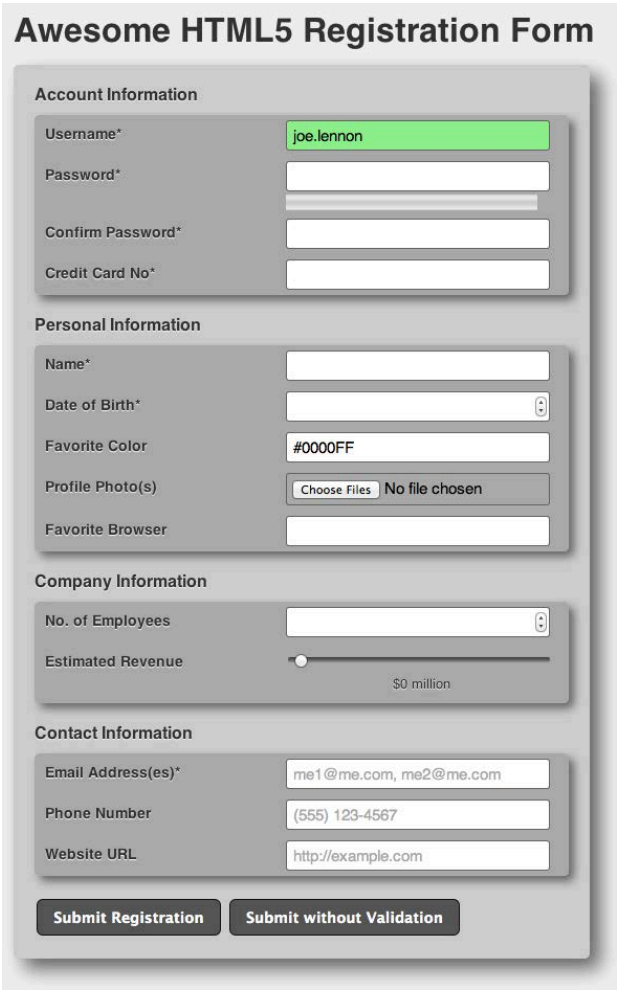

Figure 2.23. The final registration form application in action, complete with CSS3 validation styling. The background color of the field with focus represents whether it is currently valid (green) or invalid (red).

Now, having learned new techniques that will improve how your Web forms work in HTML5, and how those techniques work in compatible Web browsers, let's investigate how you can provide graceful fallbacks for those users who are using Web browsers that lack support for the new features.

2.5 Providing fallbacks for old browsers

Even though lots of users are still using old browsers, you can start implementing HTML5's new form features in your applications right now. Any browsers that don't support new input

types and attributes will simply treat them as standard input elements or ignore any unrecognized attributes. With that said, it would be much more beneficial to your users if you were to provide JavaScript fallbacks that ensured the majority of users who don't have a compatible browser are not left with crippled basic text inputs while others can use features like date pickers, color wheels, sliders and spinners.

In this section, you will learn how to use the Modernizr JavaScript library to detect support for HTML5 forms features. You will then learn how to use the jQuery UI widget library to add any missing widgets such as the date picker widget. Finally you will learn how to replicate HTML5 functionality in JavaScript using polyfills, to ensure that all visitors to your website can benefit from a similar user experience. Ready? Here we go!

2.5.1 Checking feature support with Modernizr

The Modernizr JavaScript library allows you to easily detect whether or not the Web browser a visitor is using supports certain HTML5 and CSS3 features. When downloading Modernizr from http://www.modernizr.com/download/, you can choose to download the development version, which contains tests for almost every HTML5 and CSS3 feature detectable using JavaScript, or you can download a lean, configurable production build that only contains the tests you need and is pre-shrunk to a minimal size. Which option you choose is not important, but if you do choose the production build, make sure that you at least select "Input Attributes" and "Input Types" from the HTML5 section, and "Modernizr.load (yepnope.js)" from the Extra section. Also, to load Modernizr, you should include the modernizr.js file you downloaded in the <head> section of your HTML document, following any stylesheets you are including.

New in Modernizr 2.0 is the inclusion of the yepnope.js dynamic loader library, which allows you to use Modernizr to test for feature support, and based on the result load different external CSS and JavaScript files. Say, for example, you want to use the date input type in your document, but in browsers that don't implement a date picker widget, you want to load a JavaScript version, datepicker.js. Rather than load the datepicker.js file for everyone (even those users whose browser supports a native date picker widget), you can use a Modernizr test to determine whether the library should be loaded, as follows:

```
Modernizr.load([
    {
        test: Modernizr.inputtypes.date,
        nope: ['datepicker.js']
    }
]);
```

This code will check if the user's browser supports the date input type, and if it does not, loads the datepicker.js file. Alternatively, if you want to use Modernizr to execute a block of JavaScript code rather than dynamically load a file, you can use the following syntax:

```
if(!Modernizr.inputtypes.date) {
    //Code to show datepicker
}
```

The code in the if-block will only execute if Modernizr detects that the user's browser does not support the date input type. Table 2.3 lists the different properties you need in Modernizr to detect support for HTML5 input types.

Table 2.3. Modernizr properties for HTML5 input element types

Modernizr property name	HTML5 input type
Modernizr.inputtypes.color	<input type="color">
Modernizr.inputtypes.date	<input type="date">
Modernizr.inputtypes.datetime	<input type="datetime">
Modernizr.inputtypes.datetime-local	<input type="datetime-local">
Modernizr.inputtypes.email	<input type="email">
Modernizr.inputtypes.month	<input type="month">
Modernizr.inputtypes.number	<input type="number">
Modernizr.inputtypes.range	<input type="range">
Modernizr.inputtypes.search	<input type="search">
Modernizr.inputtypes.tel	<input type="search">
Modernizr.inputtypes.time	<input type="time">
Modernizr.inputtypes.url	<input type="url">
Modernizr.inputtypes.week	<input type="week">

You can also use Modernizr to test for browser support of new HTML5 input attributes. The relevant properties for these are listed in the Table 2.4.

Table 2.4. Modernizr properties for HTML5 input element attributes

Modernizr property name	HTML5 input attribute
Modernizr.input.autocomplete	autocomplete
Modernizr.input.autofocus	autofocus
Modernizr.input.list	list

Modernizr.input.max	max
Modernizr.input.min	min
Modernizr.input.multiple	multiple
Modernzir.input.pattern	pattern
Modernzir.input.placeholder	placeholder
Modernizr.input.required	required
Modernizr.input.step	step

Now that you know how to use Modernizr to detect support for HTML5 forms features, let's see how you can use jQuery UI to provide fallbacks for those browsers that don't support these features.

2.5.2 Adding missing functionality with jQuery UI

Adding support for missing features using jQuery and its accompanying widget framework, jQuery UI, is fairly easy thanks to Modernizr. First, you can use Modernizr.load to only load jQuery and jQuery UI if the browser does not support all of the features required by your application. Then, it's a matter of checking for the features you are using individually and if they are not supported, providing some JavaScript code to use a jQuery alternative to fall back on.

Let's take an example of using jQuery UI to provide a datepicker on browsers where a native datepicker has not been provided. Given the following HTML markup:

```
<input type="date" name="date_of_birth" id="date_of_birth">
```

You can perform a Modernizr check to see if the date input type is supported by the user's Web browser, and render a jQuery UI date picker if necessary, using the following JavaScript code:

```
if(!Modernizr.inputtypes.date) $("#date_of_birth").datepicker();
```

If you test this in Opera, you will see the native HTML5 widget rendered, as you would expect. If you then try it in Firefox, you will see the jQuery UI date picker instead. When you try to run it in Chrome or Safari, you will get a combination of the weird spinbox widget that WebKit provides for date input types and the jQuery UI datepicker. The first issue here is that the date produced by the jQuery UI datepicker isn't necessarily going to be compatible with the HTML5 date input type. You can rectify this by telling the jQuery UI datepicker what format it should use for output values, as follows:

```
if(!Modernizr.inputtypes.date)
    $("#date_of_birth").datepicker({dateFormat: "yy-mm-dd"});
```

It would be much nicer if Chrome and Safari ditched those awful spinboxes altogether though, so let's use some trickery to get rid of the date field and insert a standard text field in its place, and then use jQuery UI to make this field a date picker widget.

```
if(!Modernizr.inputtypes.date) {
    var oldDateField = $("#date_of_birth");
    var newDateField = oldDateField.clone();
    newDateField.attr("type", "text");
    newDateField.insertBefore(oldDateField);
    oldDateField.remove();
    newDateField.attr("id", "date_of_birth");
    newDateField.datepicker({
        dateFormat: "yy-mm-dd"
    });
}
```

Unfortunately, you can't simply change the type attribute of the date_of_birth field dynamically, as this will cause errors in most browsers. Instead, create a clone of the original field, change the type property of the cloned field, insert it and remove the old one. Then, attach the date picker to the new field. The end result is a pretty jQuery UI date picker, shown in figure 2.24, which looks consistent in all browsers that don't support a native date picker widget.

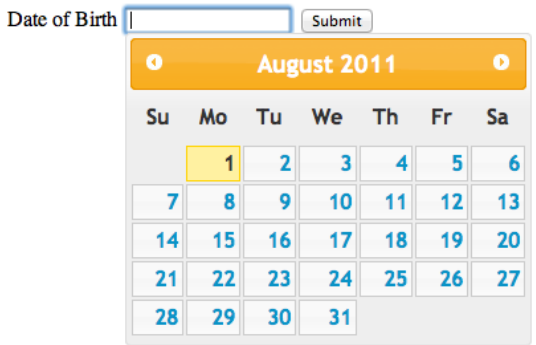

Figure 2.24. Providing the jQuery UI date picker widget as a fallback in browsers without a native date picker widget. The widget provides the same functionality as the date picker implemented in Opera. When browsers support native date pickers, they will be preferred to this jQuery one.

You can use the same technique to implement other widgets such as number spinners, sliders, progress bars and autocomplete fields where native support is not available. Of

course, you can also use this approach to use jQuery plugins to mimic functionality such as autofocus, placeholder and so forth.

2.5.3 Ensuring backward compatibility using polyfills

There are a number of projects that aim to replicate the functionality provided in HTML5 so that a standard JavaScript fallback is available for older browsers. These projects are known in the Web development community as “polyfills”, pieces of fallback code that will be executed in the event of a Web browser not supporting a particular feature natively.

There are numerous polyfills available for providing HTML5 web forms features in older browsers, including:

- H5F from Ryan Seddon, which does not rely on any third-party JavaScript libraries, available from https://github.com/ryanseddon/H5F
- jquery.html5form from Matias Mancini, an excellent HTML5 form polyfill plugin for jQuery developers, available from http://www.matiasmancini.com.ar/jquery-plugin-ajax-form-validation-html5.html
- html5Widgets from Zoltan Dulac, which provides widgets for various new form input types, cross-browser constraint validation API and much more, available from https://github.com/zoltan-dulac/html5Widgets

HTML5 polyfills are not limited to just Web forms. The GitHub project site for Modernizr includes a page filled with links to different projects that provide stop-gap functionality that can be used to provide HTML5 functionality to older browsers. You can find this page at https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills.

2.6 Summary

Beyond the shadow of a doubt, Web forms are hugely important in modern Web applications. Virtually every Web application ever made relies on Web forms in some respect to provide functionality to users. Whether for data input and retrieval, search queries, or communications and messaging, Web forms are everywhere. The support for Web forms in previous versions of HTML was pretty basic to say the least. If you wanted to provide the user with any sort of feedback, or allow them to enter data using a reasonable form of input (think of date pickers for entering dates), you had to rely on third-party JavaScript libraries to facilitate such functionality. If your user was using a browser that did not support JavaScript, you could forget about decent widgets and client-side validation.

HTML5 introduces a lot of functionality for improving Web forms. Unfortunately, despite many aspects of these new features and improvements being part of the HTML5 specification for quite some time now, browser vendors have been relatively slow to implement these features in recent versions of the browsers. The only browser vendor to implement widgets for date and color pickers is Opera, and that looks unlikely to change anytime soon. Despite this issue, one thing is for certain – all browsers will offer complete support for these new features at some point, and when that happens, you should be ready to take advantage of it

in your applications. When you consider the flexibility you can achieve by testing for feature support with Modernizr, dynamically loading third-party libraries using yepnope.js and using polyfills to bridge the gaps in older browsers, there really is no reason not to start adding HTML5 form support to your applications right now.

In the next chapter, you'll learn about how you can use HTML5 to build desktop-style applications, with features such as "contenteditable", which allow you to turn any HTML element into an editor field, drag and drop, the File API and the History API.

5

Mobile and offline Web applications

This chapter covers

- Developing HTML5 applications for mobile devices
- Storing data on the client-side
- Managing a full client-side IndexedDB database
- Enabling applications to work offline

Smartphones are everywhere these days. Not only is the technology continually improving, but the devices are also becoming more affordable. It's actually quite possible that, in the near future, every phone will be a smartphone. Wouldn't it be great if you could develop an app that would run on nearly every phone in the world? Believe it or not, this dream may not be as far-fetched as you think. More and more devices are supporting HTML5 each day, allowing developers to build one app that works on many different devices. Some of HTML5's best new features are perfectly suited to apps for mobile devices, which tend to have limited resources and are much more likely to be used offline. In HTML5, we can use features such as client-side data storage and offline application cache manifests to make awesome mobile apps that can be used on planes, trains and deserted islands. In fact, this chapter will show you how to build a sweet HTML5 mobile app of your own.

While doing so, you will learn about developing mobile Web applications using HTML5, and about the various frameworks and tools that can make the process much simpler. You will learn how to use the Web Storage APIs localStorage and sessionStorage to store simple data structures locally on the user's machine. Next, you will discover how you can use the Indexed Database API to store entire databases and complex data structures in the browser. Finally, we will introduce you to the application cache manifest, which allows you to define the resources in your application that should be cached locally so the application is usable even when the device is offline.

Throughout this chapter, you will see how to put the concepts into practice by means of a simple mobile Web application, "My Tasks". This application, which will be fully functional even when the user is offline, will allow you to create, update and delete tasks that are stored locally in the browser, and change some settings for the application display stored in localStorage. First though, before you start developing the application, let's look at how the mobile Web has changed the entire landscape for Web application development.

5.1 Going mobile with HTML5

The mobile Web is changing the face of Rich Internet Application development. According to comScore (February 2011), 27% of the population of the United States owned a smartphone in Q4 2010. Anyone with a smartphone will be familiar with the concept of an "app" on a mobile device. These are typically applications that are stored and run on the device natively, usually purchased or downloaded from a relevant store. When Apple launched the iPhone and iPod Touch App Store back in July 2008, there was a gold rush of app development, with many developers having huge success. Google responded by launching the Android Marketplace in October 2008, with support for paid applications following in 2009. Before long, every smartphone platform had its own marketplace for applications – AppWorld for BlackBerry devices, Windows Phone Marketplace for Windows Phone 7 devices and Ovi Store for Nokia Symbian-based handsets.

One of the problems with developing a native application for a mobile device is that the APIs and programming languages used for each platform can vary widely, making it difficult and time-consuming to deploy apps to multiple mobile application marketplaces. Maintaining several codebases for the same application can be frustrating, and makes code management and bug tracking a nightmare. Fortunately, most smartphones have Web browsers with good support for HTML5, meaning that, rather than developing native apps for each platform, developers can instead choose to develop for the mobile Web, targeting multiple platforms with a single application.

In this section, you'll learn how you can use HTML5 to develop impressive mobile Web applications. You'll begin by learning how to add several elements to the <head> section of your HTML documents that will enable your Web applications to be used in the native manner of mobile devices such as the iPhone, Blackberry, and Droid. Next, you will learn about mobile HTML5 frameworks that make the process of developing mobile-specific applications much easier. Following on, you will discover some frameworks that allow you to take a mobile Web application and deploy it natively to mobile handsets via marketplaces such as the iOS App Store and the Android Marketplace. Finally, you will get started with the sample application you will build in this chapter, writing the basic HTML and JavaScript code required to navigate around the Tasks application. Let's dig in.

5.1.1 Building native-like mobile Web applications

With the advent of the mobile "app", users have come to expect mobile software to behave and look a certain way. With HTML5, JavaScript and CSS3, you can build complex Web

applications that run right in the device's Web browser, without it needing to be downloaded and installed from an app marketplace. However, on some devices, the experience doesn't quite match up to the one you will experience when using a native application. For example, on iOS devices, the browser toolbars at the top and bottom will always show in your application, which takes up valuable screen real estate that could be used by your application. The top toolbar will hide as you scroll down in your app, but it will appear again if you scroll back to the top, making it difficult to implement touch-friendly scroll handling.

To get around this, there are a few tricks you can use, enabling your application to be used in "native mode". On iOS, you can use special <meta> and <link> elements to build applications that can be placed on the home screen, complete with an icon, name and splash screen. Android supports the apple-touch-icon-precomposed property to define a high-resolution icon that should be placed on the Web app's home screen bookmark. You can also choose whether the status bar at the top of the screen should be black, grey or translucent. The following code illustrates how to employ each of these techniques in your application by adding <link> and <meta> elements to your HTML5 page's <head> section:

```
<meta name="apple-mobile-web-app-capable" content="yes">
<meta name="apple-mobile-web-app-status-bar-style" content="black">
<link rel="apple-touch-icon" href="/custom_icon.png">
<link rel="apple-touch-icon-precomposed"
href="/custom_precomposed_icon.png">
<link rel="apple-touch-startup-image" href="/startup.png">
```

With this code, an Apple user can add the Web app to his home screen using the "Add to Home Screen" menu option in Mobile Safari. When the user launches the application from the resulting app icon, the app splash screen will be displayed while it loads, and the Mobile Safari toolbars will no longer be displayed. You can even detect if the user is using the app in Mobile Safari or in the "full-screen" mode, using the following JavaScript Boolean property:

```
window.navigator.standalone
```

An example of how a mobile Web app is used natively is shown in figure 5.1.

Figure 5.1. Examples of the iOS web application “Add to Home”, App Icon and Splash Screen native-style features in action. These features allow users to use your Web application as if it were a native app.

If you are developing Web applications that you intend to be run in full-screen mode as described previously, you need to be aware that there are some rather frustrating limitations when using apps in this way. First, these applications will not be able to avail of the new Nitro JavaScript engine unveiled in iOS 4.3, so they will run slower than their Mobile Safari counterparts. In addition, if your application uses the location.hash property to save state or other properties, the property will never be kept between full-screen mode sessions (you can get around this limitation by using localStorage as a fallback for state management when the user is in full-screen mode). Perhaps the most irritating issue with full-screen mode Web apps in iOS, however, is that offline application caching does not work at all, making cache manifests useless if you want your applications to be able to work offline. If this is paramount to you, keep reading, as we’ll be introducing you to some toolkits that allow you to convert your mobile HTML5 Web apps to native apps on multiple platforms with relative ease.

5.1.2 Developing mobile Web apps using mobile HTML5 frameworks

If you are an experienced Web application developer, you will probably be familiar with the various Web and JavaScript frameworks available that make your life easier when it comes to ensuring cross-browser compatibility with your code, or that reduce your workload by giving you access to a series of User Interface (UI) widgets and components. If so, you may

have been wondering if there is any such framework for mobile Web applications. Fortunately, the answer is yes, and there is a decent choice of frameworks on offer, including:

- jQuery Mobile
- Sencha Touch
- Dojo Mobile
- Jo

Although each of these frameworks differs from the others in various ways, they all provide developers with a means of building mobile Web applications that leverage HTML5 to create a native app experience. Every one of these frameworks includes a rich set of UI components that mimic native mobile UI features such as lists, navigation bars, toolbars, tab bars, form controls, carousels and more. Each also provides a data abstraction layer that makes it easier to interact with HTML5's storage features, which we will cover in great detail later in this chapter. Finally, all of these frameworks can be used in tandem with a series of frameworks for deploying mobile Web applications as native apps on various platforms. You will learn about this more in the next section.

5.1.3 Deploying mobile HTML5 apps natively

In some cases, deploying mobile applications over the mobile Web just isn't an option. Perhaps you would like to charge for your applications and want to use the hosting, payment processing and support that a native marketplace has to offer. Maybe you need to access device features, such as the camera, that are not accessible to Web applications (not yet at least). Or maybe you want to use both options to gain as many users as you possibly can. All of these are perfectly valid reasons to want to develop a native app, and the good news is that if you have already developed a mobile Web application in HTML5, you already have most of the hard work completed. As with the mobile JavaScript framework space, there are a number of players in the native Web app deployment market, including:

- PhoneGap
- Appcelerator Titanium
- Rhomobile
- appMobi

Although the implementations vary, these frameworks typically offer a series of JavaScript APIs for accessing various aspects of the device (such as the camera, device information, contacts data and more), allowing your HTML5 application to work with parts of the device traditionally only available to native applications. The frameworks also offer some means of producing native builds of HTML5 applications for the various mobile platforms. These native builds can then be packaged up and deployed to app marketplaces, just as if they were built using traditional native SDKs.

5.1.4 Building a mobile-friendly Tasks application

Throughout the rest of this chapter, you will learn about various HTML5 features that can enhance a mobile Web application. First, you need to build the basic structure of this application, so let's define what the application will actually do. The final app will be called "My Tasks", and will allow users to create and manage a simple "To Do" list of tasks that they need to complete. The following are the features that the app will include:

- Saves user preferences (name, color scheme) in localStorage
- Adds, edits and deletes tasks in an IndexedDB/WebSQL database
- Works offline using an application cache manifest

In order to facilitate these features, the application will need to have three distinct views – a "Task List", an "Add Task" form view, and a "Settings" view, which allows users to view and change their preferences. Rather than create separate HTML pages for each of these views, let's define each into a <section> element. You will then use location.hash to determine which view should be active at any given time. This will make your application feel responsive and fast. Figure 5.2 illustrates the final application in all its glory, with all of the features implemented.

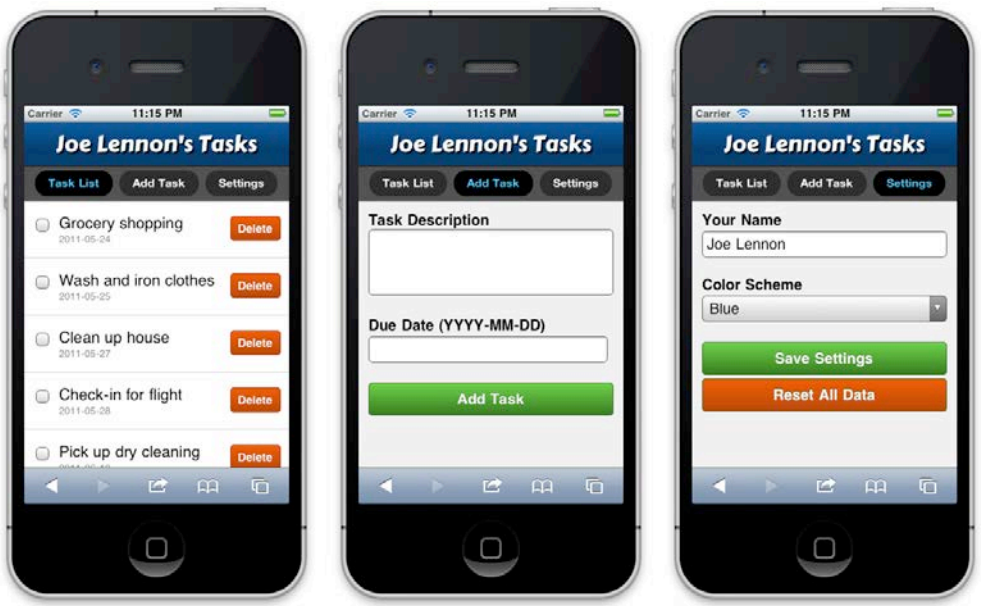

Figure 5.2. The three main views of the My Tasks app: Task List, Add Task form and Settings form. The app includes a nav bar that allows you to quickly switch between each of these views.

The application will comprise three separate files - the main HTML document, a JavaScript source file, and a CSS stylesheet file. For the purpose of simplicity, we'll keep all of these files in the same directory. Start off by defining your HTML page structure. Create a file named index.html, and add the contents of Listing 5.1 to it.

Listing 5.1. index.html – Application HTML structure

```
<!doctype html>
<html lang="en" class="blue">                                          #1
    <head>
        <meta charset="utf-8">
        <meta name="viewport" content="width=device-width,
            initial-scale=1.0, maximum-scale=1.0, user-scalable=0">
        <title>Tasks</title>
        <link rel="stylesheet"
            href="http://fonts.googleapis.com/css?family=Carter+One">   #A
        <link rel="stylesheet" href="style.css">
        <script src="app.js"></script>
    </head>
    <body class="list">                                                #2
    </body>
</html>
```

#1 class for color scheme
#A Load font from Google Font API
#2 class for active view

In this code listing you are adding a class name "blue" #1 to the <html> element. This is where the color scheme preference will be applied, and by default the blue color scheme will be used. If the user has selected a different scheme, it will be applied later using JavaScript to read the user preferences from localStorage. Similarly, you are adding a class name "list" #2 to the <body> element to indicate the active view so that you can highlight the current view on the application navigation bar.

Next, add the navigation bar by adding the code in Listing 5.2 within the <body> section of your index.html page.

Listing 5.2. index.html – Adding a Navigation Bar

```
<header>
    <h1><span id="user_name">My</span> Tasks</h1>                      #1
    <nav>
        <ul class="clearfix">
            <li><a href="#list" class="list">Task List</a></li>        #2
            <li><a href="#add" class="add">Add Task</a></li>           #2
            <li><a href="#settings" class="settings">Settings</a></li> #2
        </ul>
    </nav>
</header>
```

#1 Use ID for username
#2 Nav bar items

The main application header and navigation bar are housed inside an HTML5 <header> element. Use a with the ID “user_name” #1 to define where the user name should be put when the user sets their name using the Settings view. Then create a <ul> with three list items #2, each of which points to a hash reference for the view in question. Importantly, each of the links in these items contains a class name, which you can use to give the active view a different appearance on the screen using CSS. The final part of the HTML page is the main application sections. Insert the following code directly after the code from Listing 5.2 in index.html.

Listing 5.3. index.html – Main Application Views

```
<section class="list">
    <ul id="task_list"></ul>                                          #1
</section>
<section class="add">
    <form name="add">                                                 #2
        <label for="txt_desc">Task Description</label><br>
        <textarea id="txt_desc"></textarea><br><br>
        <label for="txt_due">Due Date (YYYY-MM-DD)</label><br>
        <input type="date" id="txt_due"><br><br>
        <input type="submit" id="btn_add" value="Add Task">
    </form>
</section>
<section class="settings">
    <form name="settings">                                            #3
        <label for="txt_name">Your Name</label><br>
        <input type="text" id="txt_name"><br><br>
        <label for="cbo_color">Color Scheme</label><br>
        <select id="cbo_color">
            <option value="blue">Blue</option>
            <option value="red">Red</option>
            <option value="green">Green</option>
        </select><br><br>
        <input type="submit" id="btn_save" value="Save Settings"><br>
        <input type="button" id="btn_reset" value="Reset All Data">   #4
    </form>
</section>
```

#1 Main task list
#2 Add Task form
#3 Settings form
#4 Clear local data

There’s nothing too complex about this HTML code. The three views are represented by three separate <section> elements. Only one of these views will ever be visible at any given time. The first section contains an empty list #1 with the id “task_list”. When you implement the IndexedDB tasks database later in this chapter, you will dynamically add tasks to this list as required. The second view contains a form #2 that will allow users to add a new task. New tasks contain a task description and a due date, and you use the new date input form type

for the latter. Finally, the third view contains another form #3, which will allow the user to enter his/her name and select a color scheme. In addition to being able to save these preferences, this view will also display a “Reset All Data” button #4 that will clear any local storage and IndexedDB data when pressed.

The application contains a substantial amount of CSS code to prettify the user interface. The majority of this code is only presentational, so we will not cover it here. The entire CSS source code for this sample application can be downloaded from the book’s website. One aspect of the CSS code that is important from a functionality point of view defines two rules that determine which view section should be visible at a given time. This code is as follows:

```
section {
    display: none;
}

body.list section.list, body.add section.add, body.settings
section.settings {
    display: block;
}
```

Without these rules, the application will have no concept of whether a particular view should be visible at any given time, so be sure to include at least these two rules in a file named style.css, stored in the same directory as index.html.

The heart of the application is the JavaScript code, which should be kept in its own file named app.js, again, stored in the same directory as the index.html file. Let’s start off by defining some simple functions that allow you to switch between the different views in your application.

Listing 5.4. app.js – Foundation JavaScript code for our application

```
var app = {
    scroll: function() {                                          #1
        window.scrollTo(0,0);
    },
    jump: function() {
        switch(location.hash) {                                   #2
            case "#add":
                document.body.className = 'add';
                break;
            case "#settings":
                document.body.className = 'settings';
                break;
            default:
                document.body.className = 'list';
        }
        setTimeout(function() {
            app.scroll();
        }, 1000);
    },
    launch: function() {
        //Enable switching between parts of application
```

```
        window.onhashchange = app.jump;                                    #A
        app.jump();                                                        #A

        //Hide address bar on Mobile Safari when orientation changes
        window.onorientationchange = app.scroll;
    }
}
window.onload = app.launch;                                                #B
```

#1 Hide nav bar on iOS
#2 Get current location.hash
#A Change views
#B Run app.launch on load

Your main application lives inside an object named app, which contains a series of functions:

- **scroll** This function will hide the address bar on iOS devices by scrolling the page #1. It doesn't need to scroll down past the top, just calling the scrollTo function is enough.
- **jump** This takes care of the application's navigation, switching to a different view based on the current URL hash #2
- **launch** This runs when the app first loads, and takes care of some initialization such as setting up event handlers and loading the correct view for first launch.

If you run the application in any HTML5 compatible Web browser, you should see that you can navigate between the different views of the application, and that the current view is highlighted in the navigation bar. In the next section, you'll learn about the Web Storage APIs defined in HTML5, and you will use this knowledge to implement the Settings view, allowing the user to define his/her name and select a color scheme, as well as resetting the settings if required.

5.2 Client-side data storage with localStorage and sessionStorage

One of the problems with traditional Web application development is that virtually all data is stored on the server-side in some form of data store, be it a database or simple file structure. Any data that is stored on the server-side must be retrieved using an HTTP request, which results in applications being less responsive than their desktop counterparts, as the user is often left waiting while the application is making the request, listening for a response, parsing that response and then doing something with that data. While certain types of data will always need to be stored on the server-side, there are many cases when the data in question is straightforward and insensitive, and can easily be stored on the client-side. That is where the HTML5 Web Storage APIs, localStorage and sessionStorage, come in. These APIs allow you to store and retrieve basic data structures in a client-side data store. In this section, you will learn how to use these APIs, how to listen for related events, and how to build these features into this chapter's sample application.

5.2.1 The need for persistent client-side data storage

Although the Web Storage API is a new feature in HTML5, web applications have been using client-side data storage for a long time in the form of cookies. Cookies allow you to store small pieces of data on the client, and are used in virtually every HTML-based Web application that uses some form of authentication or login system. Unfortunately, cookies have drawbacks that make them an unattractive prospect for storing data locally. They can only contain a maximum of 4 kilobytes of data, making them useless for persisting large amounts of data. In fact, it's just as well that cookies are restricted to 4 KB, because they are included with every HTTP request your application makes, slowing the app down and potentially opening a security hole in the process.

The Web Storage APIs in HTML5 provide a standardized API for storing and retrieving data locally in the client browser. Unlike cookies, you can store much larger quantities of data (5 MB is the standard maximum in most browsers), and the data is not transmitted in each HTTP request, ensuring that the performance of your application is not burdened by storing it. In addition, the APIs defined in the specification are extremely simple and thus easy to use. The specification also defines an event that is fired any time the Storage object changes, making it easier for developers to listen for changes to client-side data.

In the next section, you will learn how to use the localStorage and sessionStorage APIs.

5.2.2 The localStorage and sessionStorage API

The HTML5 Web Storage specification defines two APIs for storing data locally on the client – localStorage and sessionStorage. The localStorage API allows you to store data that will persist on the client machine between sessions. The data can only be overwritten or erased by the application itself, or by the user manually performing a clear down of their local storage area. The sessionStorage API is identical to the localStorage API, with the only difference being that the data stored will not persist between browser sessions, so if the user closes the browser, the data is immediately erased.

Before using these APIs, it is wise to check that the client's browser supports them. This is very straightforward, as illustrated by the following code snippet:

```
if('localStorage' in window) {
    ...
}

if('sessionStorage' in window) {
    ...
}
```

The API itself is actually very straightforward as well. To store data, use the setItem function, passing a key and a value that you want to be stored. An example of this might be:

```
localStorage.setItem("name", "Joe");
```

Alternatively, you can use the square-bracket syntax for JavaScript objects, as follows:

```
localStorage["name"] = "Joe";
```

A third alternative is to use dot-notation:

```
localStorage.name = "Joe";
```

> **WARNING** Although all three of the methods described will work, you cannot use dot-notation to create items with a key the same as a reserved JavaScript word such as "null". As a result, it's safer to always use the setItem function or the square-bracket syntax instead.

The value of a localStorage data property can be any valid JavaScript type, in other words, a string, integer, float, Boolean or even an Object. The actual value will be stored as a string, however, so if you need to store other data types, you will need to convert the string into the relevant type after you have retrieved it from Storage.

> **WARNING** It's also important to note that if a key already exists, it will be overwritten without notice or warning, so be careful!

Retrieving data from localStorage is just as straightforward:

```
var name = localStorage.getItem("name");
```

This assigns the value "Joe" to the name variable. Again, you can also use the square-bracket or dot notation syntaxes if you prefer, so the following will give the same result:

```
var name = localStorage["name"];
var name = localStorage.name;
```

If you need to remove a property, again it's all very simple, just use the removeItem function:

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

If you want to remove all localStorage properties at once, you can use the clear function:

```
localStorage.clear();
```

If you want to retrieve all objects in localStorage, you can use the length property and the key function to iterate over the localStorage object:

```
for(var i=0, len=localStorage.length; i<len; i++) {
    var key = localStorage.key(i);
    var value = localStorage[key];
    console.log(key + " => " + value);
}
```

As you can see, the localStorage API is not difficult to use. If you want to store data that will only live for the duration of a user's session, you should use the sessionStorage API instead. You can use the sessionStorage API the same way as you use the localStorage API, merely substitute any reference to localStorage in this section with sessionStorage. In the next section, you'll learn how you can listen for changes to the Storage object with an event handler.

5.2.3 Handling the storage event

Anytime the Storage object (where all the data stored in localStorage and sessionStorage is kept) changes in another browser window or tab that is accessing a page on the same domain, the "storage" event will be fired. The callback function you specify to handle the event will receive a StorageEvent object as an argument, which contains various properties about the change that was made to the Storage object. Take the following example, where you have a function that logs the StorageEvent object to the console when the "storage" event is fired.

Listing 5.5 storage.js – Handling changes to the Storage object

```
function logStorage(e) {
    console.dir(e);
}

function attachEvents() {
    if(window.addEventListener) {
        window.addEventListener("storage", logStorage, false);
    } else {
        window.attachEvent("onstorage", logStorage);
    }
}

if(window.addEventListener) {
    window.addEventListener("load", attachEvents, false);
} else {
    window.attachEvent("onload", attachEvents);
}
```

If you open this application in two separate windows of the same browser, and in one window you enter the following in the console (assuming you are using Firebug, Chrome Web Inspector, Opera Dragonfly or a similar tool), then nothing will happen in that window's console

```
localStorage.setItem("uniqueId", "100509");
```

If you check the other browser window, you should see output similar to figure 5.3.

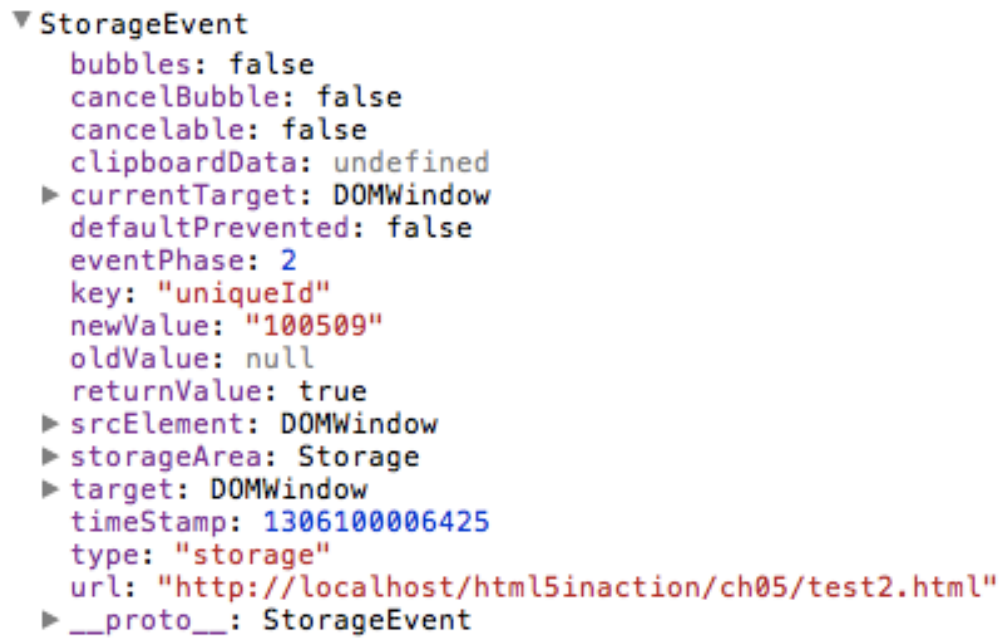

Figure 5.3. The localStorage StorageEvent object contains useful information about the item that was changed, such as the key, the previous value and new value of the item.

The five properties of note in this object are:

- key –The name of the key that has been modified
- newValue – The new value (null if the item is deleted)
- oldValue – The previous value (null if the item is new)
- storageArea – The Storage object that was affected by the change
- url – The address of the document whose key changed

You cannot cancel the storage event, as it is merely used for notification purposes. In the next section, you will learn how to take the localStorage API and use it in the Tasks application to facilitate saving user preferences.

5.2.4 Adding localStorage to the Tasks application

In the sample “My Tasks” application, you will use the localStorage API to enable the user to save app preferences, such as name, which will be displayed in the title of the application, and to change the color scheme between blue (default), red and green. If you re-open the

app.js file from earlier in the chapter, you'll need to add the functions from the following listing to the app object.

Listing 5.6. app.js – Adding settings management functions

```
loadSettings: function() {
    this.username = localStorage.getItem("name");                          #1
    this.colorScheme = localStorage.getItem("color");                      #1

    if(this.username) {
       document.getElementById("user_name").innerHTML = this.username+"'s";
#2
       document.getElementById("txt_name").value = this.username;
    } else {
       document.getElementById("user_name").innerHTML = "My";
       document.getElementById("txt_name").value = "";
    }

    if(this.colorScheme) {
       document.documentElement.className = this.colorScheme;              #3
       document.getElementById("cbo_color").value = this.colorScheme;
    } else {
       document.documentElement.className = "blue";
       document.getElementById("cbo_color").value = "blue";
    }
},
saveSettings: function() {
    var name = document.getElementById("txt_name").value;                  #4
    var colorScheme = document.getElementById("cbo_color").value;          #4

    localStorage.setItem("name", name);                                    #5
    localStorage.setItem("color", colorScheme);                            #5

    document.getElementById("user_name").innerHTML = name+"'s";
    document.documentElement.className = colorScheme;

    alert("Settings saved successfully", "Settings saved");
    location.hash = "#list";                                               #A
},
resetData: function() {
    if(confirm("This will reset settings and delete all tasks. Are you
sure?", "Confirm reset")) {
       localStorage.clear();                                               #6
    }
}
```

#1 Retrieve item
#2 Set heading to name
#3 Set <html> class name to color
#4 Get form values
#5 Save item
#A Redirect to task list
#6 Clear data

In the loadSettings function, you are retrieving the values of the name and color properties from localStorage using the getItem function #1, and assuming these values are actually set, you then set the heading span with the ID "user_name" to the value of the name property #2, and set the className of the root document element (the <html> element) to the value of the color property #3. In the saveSettings function, you first read the form values for the name and color scheme fields #4, before using the setItem function to save this data to localStorage #5. Finally, the resetData function uses the clear function to remove all data from localStorage #6. In order to actually use these functions, you also need to add some code to your app's launch function, as shown in the following listing.

Listing 5.7. app.js – Adding event handlers to the launch function

```
app.loadSettings();                                                    #1
document.forms.settings.onsubmit = app.saveSettings;                   #2
document.getElementById("btn_reset").onclick = app.resetData;          #3
```

#1 Retrieve settings
#2 Save on form submit
#3 Reset event handler

First, you load the settings using the loadSettings function when the app starts #1, before attaching two event handlers, one to the submit event of the settings form #2, and the second to the reset button, which should clear down the local storage data #3. If you launch the application in a compatible browser, you should be able to navigate to the Settings view, and set the name and color scheme to the values of your choice. When you submit the form, the heading and color scheme should update accordingly, and you should be taken to the (empty) task list page. If you then press the reset button on the settings page, you should see that the settings have been cleared down and the app reverts to the default settings. Because you are using localStorage, these settings will persist between browser sessions, unless the user specifically clears down their localStorage area via the browser preferences screen. Go ahead and test it out. You should be able to refresh the page, restart your browser and even restart your computer, and the data should still persist. Pretty neat huh?

In the next section, we'll show you how to take things even further with client-side data storage using the Indexed Database API and the now defunct Web SQL specification. We'll also show you how to add the real meat to your sample application, implementing the actual functionality that allows you to add, edit and delete tasks.

5.3 Managing more complex data with IndexedDB

In the previous section, you learned about how you can use localStorage and sessionStorage to store basic data structures on the client-side. This is perfect for storing small and simple data such as user preferences or session-related information. If you want to store more complex data structures, and you want this data to be easily searchable using indexes, those APIs are just not going to cut it. Fortunately, HTML5 specifies an additional client-side API for data storage for this exact purpose, the Indexed Database API, or IndexedDB for short. In

this section, you will learn about how IndexedDB came about, how it works and how to do basic storage and retrieval using the API. You will also learn how to use database cursors to iterate over your data records, and how to create indexes that will speed up your queries. At the end of this section you will learn how to add IndexedDB support with a Web SQL fallback to the “My Tasks” sample application.

5.3.1 The database that lives in your Web browser

IndexedDB is a specification for providing a client-side API to a transactional database that is stored on the client rather than the server. Whereas localStorage and sessionStorage are useful for storing simple key/value pairs, IndexedDB supports much more advanced functionality, including in-order retrieval of keys, support for duplicate values and efficient value searching using indexes. The specification requires supporting browsers to provide a transactional database system that allows keys to be associated with one or more values, allowing keys to be traversed in a deterministic order, typically using B-tree data structures that keep the data sorted and enable sequential access and searching. In short, this means that IndexedDB is much more suitable than the likes of localStorage when it comes to storing numerous records of data, and providing an efficient means of retrieving and filtering this data when required.

Up until recently, the HTML specification looked to be going down a different path for a client-side database API. The original plan was to implement Web SQL, a client-side implementation of a database that supports the Structured Query Language (SQL) as a query interface. In fact, several browser vendors had already implemented the Web SQL specification in recent versions of their browsers. You can use Web SQL in any recent version of Safari, Chrome, Opera, Mobile Safari on iOS devices and the Android browser. Neither Firefox nor Internet Explorer added support for Web SQL, and with all current implementations using SQLite at the back end, the W3C felt that there were not enough independent implementations of the standard and started to look at alternatives.

Mozilla’s objection to using SQL as the API was centered on the lack of a clear standard for the language, with their preference to define a new standard JavaScript API rather than trying to define a subset of SQL for the Web. There was also the concern that by relying on SQLite, the implementation of Web SQL could become too integrated with the SQLite database itself. In fact, the people at Mozilla did not have any concern with using SQLite as the actual database behind the scenes, but were not happy to use it as the API to interface with data from JavaScript applications.

In November 2010, the W3C stopped working on the Web SQL Database specification, instead concentrating their efforts on the Indexed Database API spec. Although this is probably a good thing from the point of view that developers will now get access to a full JavaScript API for client-side databases, it also means that the current level of browser support has taken a step back, meaning that developers will likely need to include fallbacks for other browsers until support for IndexedDB is widespread. Later in this section, you will learn how to add IndexedDB features to the “My Tasks” application, but you will also add

Web SQL features as a fallback for those browsers that have yet to include an implementation of IndexedDB. Firefox 4 and Chrome 11 both support the new API, and hopefully other browser vendors will soon follow suit. First though, we'll get you started with using the IndexedDB API, learning about how to create databases and object stores, and how to get your head around the asynchronous nature of the API.

5.3.2 Getting started with the IndexedDB API

Because IndexedDB was a late addition to the HTML5 specification, it suffers from a lack of widespread browser support. As a result, you will need to ensure that your visitor's browser can use IndexedDB, providing a fallback in case it does not. According to the specification, you should be able to do this with the following code:

```
if('indexedDB' in window) {
    ...
}
```

Right now, only Google Chrome and Mozilla Firefox have implemented IndexedDB, and both have chosen to use their own prefixes on the object name for now. As a result, you'll need to check for both of these object names also. If you define the following code at the top of your application's JavaScript, you can then use the previous if statement to check for support.

```
if(!('indexedDB' in window) && ('mozIndexedDB' in window ||
  'webkitIndexedDB' in window)) {
    window.indexedDB = window.mozIndexedDB || window.webkitIndexedDB;
    window.IDBTransaction = window.IDBTransaction ||
        window.webkitIDBTransaction;
    window.IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;
}
```

IndexedDB uses an asynchronous API, which means that it won't block your code while it is performing a particular function. As a result, rather than writing code in a linear fashion, you will need to use callback functions that ensure that the database request has either completed successfully or failed in the process. This asynchronous approach makes it difficult to try out the API using a console in a tool like Firebug, Dragonfly or the WebKit Web Inspector, so it's usually better to put this code into a Web page and try it out that way instead. The concept is better explained by an example, so let's take a look at the API you use to open a database.

```
var db;
var req = window.indexedDB.open("MyDB", "Database Description");
req.onsuccess = function(e) {
    db = e.target.result;
}
req.onfailure = function(e) {
    logError(e);
}
```

With this code, you define a request that opens a connection to a database named “MyDB”, creating it if it doesn’t already exist. Then, you add a callback function to the request, which gets a handle to the database if it has been successfully opened. Next, define a callback function to handle an error if one occurs in the process. Assuming the transaction is successful, the db variable should contain an IDBDatabase object, which you can use to create and manipulate object stores in the database.

All data in an IndexedDB database is stored inside an object store. Each database can contain many object stores, which can be roughly thought of as equivalent to a table in a relational database management system (RDBMS). In turn, each object store comprises a collection of zero or more objects, the equivalent of rows in a RDBMS. Figure 5.4 illustrates the structure of an IndexedDB database.

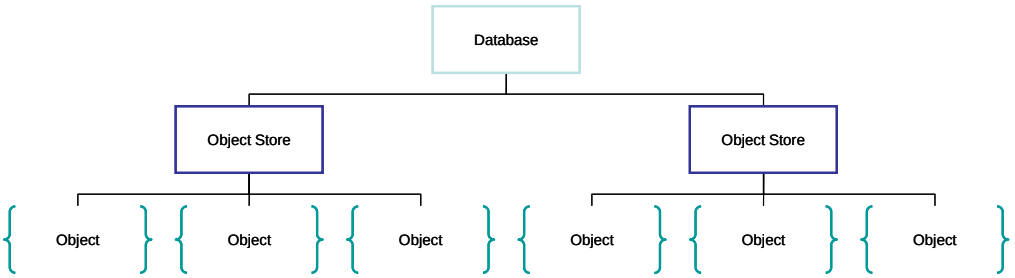

Figure 5.4. Hierarchical structure of an IndexedDB database. Each database can have many object stores, which themselves can contain many objects. The object is the structure for a data record, equivalent to a row in a relational database.

Another point: You can only create and delete object stores in IndexedDB in the scope of a setVersion API transaction. Again, the easiest way to explain is by showing you some code.

```
if(db.version != "1") {
    req = db.setVersion("1");
    req.onsuccess = function(e) {
        var objStore = db.createObjectStore("MyStore", { keyPath: "id" });
    }
}
```

First, you check the database version to determine whether you need to create the object store before actually initiating the setVersion request. Then define a callback function that will create an object store named “MyStore” when the setVersion process completes successfully. The second argument in the createObjectStore function allows you to define options; in this case you are setting the key path of the store to “id”. Optionally, you could also specify the autoIncrement property to automatically increment the key value using a key generator. This option property is a Boolean, and it defaults to false. The

createObjectStore function returns the object store immediately and does not need any callback functions to be defined to assign the object store to a variable.

If you need to get a handle to an existing object store, you can do so using the objectStore function of the transaction object as follows:

```
var tx = tasksDB.db.transaction(["MyDB"], IDBTransaction.READ_WRITE, 0);
var objStore = tx.objectStore("MyStore");
```

That code will open a read/write transaction object, which is then used to get a handle to the object store. In the next section, you'll learn how to use several APIs to add, update, delete and retrieve data from an object store.

5.3.3 Performing CRUD and basic data retrieval with transactions

A database is pretty much useless unless it has some data stored in it, so let's move on and actually start using some create, read, update and delete (CRUD) transactions to work with data in the object store. To insert data, you must get a handle to the object store you want to add data to and then use the add function to insert an object, as follows:

```
var person = { id: 1, name: "Joe", age: 26 }
var req = objStore.add(person);
req.onsuccess = function() {
    alert("Person added successfully", "Success");
}
```

As is plain to see, you simply pass the object you want to add to the store as an argument to the add method of that store. You can then attach a callback function to the onsuccess property to display a message when the transaction has completed. To update data, you can use the put method, which works in the exact same way. You must specify the key for the object you want to overwrite or it will just create a new object instead. If you wanted to update the person object from the previous example, you could simply use the following code:

```
var person = { id: 1, name: "John", age: 30 }
var req = objStore.put(person);
req.onsuccess = function() {
    alert("Person updated successfully", "Success");
}
```

To delete an object from the store, use the delete method, passing the key value as an argument. One gotcha to watch out for with this method, however, is that some browsers will balk at the sight of a method named delete, causing an exception be raised. This is due to the fact that delete is a reserved keyword in JavaScript. Fortunately, you can get around this by using the square bracket notation for the delete method instead. So to delete the person object created in the earlier example, you would use the following:

```
var id = 1;
var req = objStore['delete'](id);
req.onsuccess = function() {
    alert("Person deleted successfully", "Success");
}
```

Rather than delete objects individually, you can also clear down the entire store using the clear method. This API is very straightforward, and requires no arguments to be passed to it.

```
var req = objStore.clear();
req.onsuccess = function() {
    alert("Store cleared successfully", "Success");
}
```

You can also use the object store handle to retrieve data from the store using its key. You can do this using the get method.

```
var id = 1;
var req = objStore.get(id);
req.onsuccess = function(e) {
    alert("name is "+e.result.name+" and age is "+e.result.age);
}
```

The get method is useful if you are storing basic data structures in key/value pairs, but to be honest, if this is what you are thinking of using IndexedDB for then you should probably just use localStorage instead, as the API is much simpler and doesn't require you to use asynchronous callbacks. In the next section you'll learn how to use cursors to query, filter and iterate over data sets retrieved from object stores in IndexedDB.

5.3.4 Using cursors to retrieve records

Cursor is a generic term used to describe a control structure in a database that allows you to iterate through the records stored in it. Cursors typically enable you to filter out records based on certain characteristics, and to define the order in which the result set is returned. You can then sequentially move through the record set returned by the cursor, retrieving the data for use in your applications. Cursors in IndexedDB allow you to traverse over a result set that is defined by a key range, moving in a direction of either an increasing or decreasing order of keys.

The basic usage of a cursor in IndexedDB will retrieve all of the records in an object store. Like most aspects of the API, cursor methods are asynchronous, and you must define callback functions in order to interact with the data that is returned by them. Let's take a look at an example of a cursor that retrieves all records using the openCursor API.

Listing 5.8. Using a cursor to traverse through records in an IndexedDB object store

```
var cursor = objStore.openCursor();                                    #A
```

```
cursor.onsuccess = function(e) {
    var result = e.target.result;
    if(result == null) return;
    console.dir(result.value);                                    #B
    result['continue']();                                         #1
}
```

#A Asynchronously retrieve records
#B Output current object
#1 Continue to next record

The callback function will be called for each call to the continue function #1 of the result object, unless it is null in which case it will exit. If you try to use the continue function using dot notation it will cause exceptions in some Web browsers, so you again must use the square bracket notation instead. You can filter the result set that is returned by a cursor using a key range. To do this, you must first define a key range using the bound method of the IDBKeyRange object. Then, pass the key range into the openCursor API to tell IndexedDB to filter the result set to only those keys that in the specified range.

```
var keyRange = new IDBKeyRange.bound(1, 10, true, true);
var cursor = objStore.openCursor(keyRange);
cursor.onsuccess = function(e) {
    var result = event.target.result;
    if(result == null) return;
    console.dir(result.value);
    result['continue']();
}
```

This technique is perhaps most useful if you want to use pagination in your application and are using numeric keys. The first and second arguments define the lower and upper limits for the keys to be returned, and the third and fourth arguments define whether the lower and upper limit values, respectively, should they be included in the range or not. Of course, you are quite likely to want to filter your result sets by values other than the actual key. That is where indexes come into play, which, conveniently, we cover next.

5.3.5 Working with indexes

To traverse through a set of records that is filtered on a data property other than the key, you must create an index. An index in IndexedDB (this could get confusing!) can be thought of like a secondary key in the object, which can be used to filter result sets. Indexes can only be created in the scope of a database version change transaction, similar to object stores. In fact, because an object store is immediately returned when created (it is not asynchronous), it makes the most sense to create indexes immediately after you create the object store itself. Take the following example, where you define an index on the “name” property.

```
if(db.version != "1") {
    req = db.setVersion("1");
```

```
        req.onsuccess = function(e) {
            var objStore = db.createObjectStore("MyStore", { keyPath: "id" });
            var nameIndex = objStore.createIndex("nameIndex", "name", true);
        }
    }
```

Indexes are also returned immediately when created, so you do not need to add an asynchronous callback function. If you need to get a handle to an existing index in your code, you can do so using the index method on the object store:

```
var nameIndex = objStore.index("nameIndex");
```

To traverse over the result set of an index, you use a cursor just as you would on an entire object store. As with object store cursors, you can limit the result set using a key range, except in the case of an index, the key in question will be the one you have defined in your index (the “name” property in the previous example). If you use the openCursor API, the value returned will be the entire object; alternatively, you can use the openKeyCursor API, which will only return the value of the object’s main key (“id” in our example).

In the next section, you’ll take the concepts you have learned about IndexedDB in this chapter and apply them to the sample application “My Tasks”. You will open a database connection and create an object store, adding, updating and deleting records as required. You will also iterate over the task objects in the store using a cursor. In addition to using IndexedDB, the code presented in the next section will also provide a fallback for those HTML5 browsers that have implemented the Web SQL specification but have yet to provide support for IndexedDB.

5.4 Adding IndexedDB functionality to the Tasks application

The IndexedDB API can be notoriously complex at first glance, particularly if you do not have experience with writing JavaScript code that works asynchronously. The best way to learn how to use it is through an example, and in this section you will do just that, adding database functionality to the “My Tasks” application you’ve been building throughout this chapter.

IndexedDB was added to HTML5 quite late in the specification process. As a result, browser support for it has been much slower than with other parts of the specification. Before IndexedDB, HTML5 included a client-side database specification known as Web SQL, which defined an API for a full relational database that would live in the browser. Although Web SQL is now dead and is no longer part of HTML5, many browser vendors had already provided decent support for it, particularly mobile browsers. As a result, we will use Web SQL as a fallback in our sample app, ensuring that it runs on Web SQL compatible browsers, even if they do not yet support IndexedDB. As IndexedDB support becomes more widespread, our app will automatically choose to use it over Web SQL where available.

In this section, you will use the IndexedDB API to create, connect to and use a database that is stored on the client-side in the user's Web browser. You will implement the "Add Task" form view in the application, storing new tasks in the database. Next, you will update existing tasks, allowing users to mark tasks as "complete" by checking the relevant checkbox. You will also enable users to delete tasks, either one at a time by clicking on a single task's Delete button, or all at once using the "Clear Tasks" feature. Finally, you will load existing tasks from IndexedDB and display them to the user, complete with the UI features required to allow them to update and delete the task if required.

5.4.1 Connecting to the database

In order to add IndexedDB and Web SQL functionality to the "My Tasks" app, you will need to add the code from the forthcoming listings to the app.js external JavaScript file. The majority of the functions you add will be added to a new tasksDB object, which will live alongside the main app object you created earlier. Let's start off by doing some name mapping for prefixed implementations in Firefox and Chrome, and defining our basic tasksDB object shell.

Listing 5.9. app.js – The shell for our data-related functions

```
if('mozIndexedDB' in window || 'webkitIndexedDB' in window) {
    window.indexedDB = window.mozIndexedDB || window.webkitIndexedDB;
    window.IDBTransaction = window.IDBTransaction ||
      window.webkitIDBTransaction;
    window.IDBKeyRange = window.IDBKeyRange || window.webkitIDBKeyRange;
}

var tasksDB = {
    db: null
};
```

Let's jump right in and start adding functions to the tasksDB object. The first function you need to define is open, which will be responsible for opening a connection to either an IndexedDB or Web SQL database, creating an object store or table if required, and firing the load function to retrieve any existing tasks. Add the code from the following listing inside the tasksDB object defined in Listing 5.8 previously.

Listing 5.10. app.js – The open function

```
open: function() {
    if('indexedDB' in window) {
        var req = window.indexedDB.open("tasks");                          #1
        req.onsuccess = function(event) {
            tasksDB.db = event.target.result;                              #2
            if(tasksDB.db.version != "1") {
                req = tasksDB.db.setVersion("1");
                req.onsuccess = function(e) {
                    var objStore = tasksDB.db.createObjectStore(           #3
                        "tasks", { keyPath: "id" });                       #3
```

```
                    tasksDB.load();
                }
            } else {
                tasksDB.load();
            }
        }
    } else if('openDatabase' in window) {
        tasksDB.db = openDatabase('tasks', '1.0', 'Tasks database',
            (5*1024*1024));
        tasksDB.db.transaction(function(tx) {                          #4
            tx.executeSql('CREATE TABLE IF NOT EXISTS tasks ('
                + 'id INTEGER PRIMARY KEY ASC, desc TEXT, due DATETIME,'
                + 'complete BOOLEAN)', [],
                tasksDB.onsuccess, tasksDB.onerror);
        });
        tasksDB.load();
    }
}
```

#1 Open database
#2 Connect to database
#3 Create object store
#4 Start SQL transaction

This might seem like a lot of code to open a connection to a database, but it also takes care of creating an object store if necessary, and provides a Web SQL fallback for browsers that don't support IndexedDB. The function checks if the user agent support can use IndexedDB; if it can, it will immediately attempt to open a connection to a database named "tasks" #1. As IndexedDB APIs are asynchronous, define a callback function that allows you to get a handle to the database object #2, which you assign to the tasksDB.db property. You can use this property in other functions later for convenience. After checking the version number of the database, call the setVersion function if you need to create an object store, which must be done in the scope of a setVersion transaction. You then actually create the object store itself #3, giving it the name "tasks" and setting the key property as "id".

As previously mentioned, in case IndexedDB is not supported (as it is not in current versions of many desktop and mobile Web browsers), you can provide a Web SQL fallback, which will allow the application to work in those browsers. Keep in mind that there are many older browsers (and even not-so-old versions of Internet Explorer) that won't support any of these new HTML5 features, so be sure to use a relatively recent browser if you expect this application to work. The openDatabase API allows you to get a connection to a Web SQL database. In this case, you give it the name "tasks", a version of "1.0", a description and a maximum size of 5 megabytes. Then, use a transaction to issue an executeSql command #4, which will create a table named "tasks" if it does not already exist.

5.4.2 Adding new tasks to the database

Next, let's add the code which will allow users to add a new task. Again in the tasksDB object, add two new functions, addTask and addSuccess, as outlined in the following listing.

Listing 5.11. app.js – Adding tasks

```
addTask: function() {
    var task = {
        id: new Date().getTime(),
        desc: document.getElementById("txt_desc").value,
        due: document.getElementById("txt_due").value,
        complete: false
    }
    if('indexedDB' in window) {
        var tx = tasksDB.db.transaction(["tasks"],                          #1
            IDBTransaction.READ_WRITE, 0);
        var objStore = tx.objectStore("tasks");
        var req = objStore.add(task);                                       #2
        req.onsuccess = tasksDB.addSuccess;
        req.onerror = tasksDB.onerror;
    } else if('openDatabase' in window) {
        tasksDB.db.transaction(function(tx) {
            tx.executeSql('INSERT INTO tasks(desc, due, complete) VALUES('
                '?, ?, ?)',
                [task.desc, task.due, task.complete],
                tasksDB.addSuccess, tasksDB.onerror);
        });
    }

    return false;
},
addSuccess: function() {
    tasksDB.load();                                                         #3
    alert("Your task was successfully added", "Task added");
    document.getElementById("txt_desc").value = "";
    document.getElementById("txt_due").value = "";
    location.hash = "#list";                                                #4
}
```

#1 Initiate transaction
#2 Add task
#3 Reload tasks
#4 Redirect to Task List

In the addTask function, you first build up a task object, and then initiate a database transaction #1. Then, you call the add API on the object store to save the task object #2, defining a callback function tasksDB.addSuccess, which will be fired when the object has been successfully added to the object store. Again, perform a similar fallback Web SQL operation if IndexedDB is not available. The addSuccess function reloads the tasks from the database #3 and displays a message, resetting form fields and then redirects the user to the main task list view #4 so they can see their new task amongst their other tasks. The final piece of the puzzle with regards to the “Add Task” feature is to connect the new tasksDB.addTask function up to the relevant form. You also need to open the connection to the database at application launch. In the app.launch function you created earlier in this chapter, add the following code to the end of the function:

```
document.forms.add.onsubmit = tasksDB.addTask;
tasksDB.open();
```

5.4.3 Modifying existing tasks

Updating, deleting and clearing tasks works in a similar manner. You will create functions updateTask and deleteTask, with a drop and dropSuccess callback combination used for the clearing tasks functionality. The code for all of these functions is as follows.

Listing 5.12. app.js – Updating, deleting and clearing tasks

```
updateTask: function(task) {
    if('indexedDB' in window) {
        var tx = tasksDB.db.transaction(["tasks"],
            IDBTransaction.READ_WRITE, 0);
        var objStore = tx.objectStore("tasks");
        var req = objStore.put(task);                                    #1
        req.onerror = tasksDB.onerror;
    } else if('openDatabase' in window) {
        var complete = 0;
        if(task.complete) complete = 1;
        tasksDB.db.transaction(function(tx) {
           tx.executeSql('UPDATE tasks SET complete = ? WHERE id = ?',
                [complete, task.id],
                tasksDB.load, tasksDB.onerror);
        });
    }
},
deleteTask: function(id) {
    if('indexedDB' in window) {
        var tx = tasksDB.db.transaction(["tasks"],
            IDBTransaction.READ_WRITE, 0);
        var objStore = tx.objectStore("tasks");
        var req = objStore['delete'](id);                                #2
        req.onsuccess = function(event) {
            tasksDB.load();
        }
        req.onerror = tasksDB.onerror;
    } else if('openDatabase' in window) {
        tasksDB.db.transaction(function(tx) {
            tx.executeSql('DELETE FROM tasks WHERE id = ?', [id],
                tasksDB.load, tasksDB.onerror);
        });
    }
},
drop: function() {
    if('indexedDB' in window) {
        var tx = tasksDB.db.transaction(["tasks"],
            IDBTransaction.READ_WRITE, 0);
        var objStore = tx.objectStore("tasks");
        var req = objStore.clear();                                      #3
        req.onsuccess = tasksDB.dropSuccess,
        req.onerror = tasksDB.onerror;
    } else if('openDatabase' in window) {
```

```
            tasksDB.db.transaction(function(tx) {
               tx.executeSql('DELETE FROM tasks', [], tasksDB.dropSuccess,
                   tasksDB.onerror);
            });
        }
    },
    dropSuccess: function() {
        tasksDB.load();
        app.loadSettings();
        alert("Settings and tasks reset successfully", "Reset complete");
        location.hash = "#list";
    }
```

#1 Update task
#2 Delete task
#3 Clear object store

Much of the code in these functions is similar to the addTask function – you typically need to create a transaction object, get a reference to the object store and then call the relevant API on that reference. In the updateTask function, you use the put API #1 with the task object as an argument to update the task in the database. It is imperative that the task object has the correct key value, or it may create a new object in the store rather than update the existing one. For deleting tasks, use the delete API function. Some browsers don't like this as delete is a reserved word in JavaScript, so to be safe you should use the square bracket notation to call the API #2. Clearing the object store is as simple as calling the clear API #3. In this case, attach a callback function to the success event on this API, which will reload the tasks from the database (should now be empty), reload settings from localStorage and redirect the user to the task list view.

Back in the app object, you should add the following line to the resetData function, just below the line that clears the localStorage settings data:

```
    tasksDB.drop();
```

5.4.4 Loading tasks from the database

The final piece of functionality that you need to add to your application is actually loading tasks themselves into the "Task List" view, and creating event listeners that allow the user to update the task by checking the box to indicate it is completed, and to delete the task by tapping the red button that appears on the right side of each task. This code is the heart of your application, as it determines how the tasks should be displayed and it connects up the relevant events as required. These functions should be added to the tasksDB object like the majority of the rest of the code discussed in this section.

Listing 5.13. app.js – Loading and displaying tasks

```
    load: function() {
        var task_list = document.getElementById("task_list");
        task_list.innerHTML = "";
```

```
    if('indexedDB' in window) {
    var tx = tasksDB.db.transaction(["tasks"], IDBTransaction.READ_WRITE,
      0);
        var objStore = tx.objectStore("tasks");
        var cursor = objStore.openCursor();                              #1
        var i = 0;
        cursor.onsuccess = function(event) {
            var result = event.target.result;
            if(result == null) return;
            i++;
            tasksDB.showTask(result.value, task_list);                   #2
            result['continue']();
        }
        tx.oncomplete = function(event) {
            if(i === 0) {
               var emptyItem = document.createElement("li");
               emptyItem.innerHTML = '<div class="item_title">'
                    +'No tasks to display. <a href="#add">Add one</a>?';
               document.getElementById("task_list").appendChild(emptyItem);
            }
        }
    } else if('openDatabase' in window) {
        tasksDB.db.transaction(function(tx) {
           tx.executeSql('SELECT * FROM tasks', [], function(tx, results) {
                var i = 0, len = results.rows.length,
                    list = document.getElementById("task_list");
                for(;i<len;i++) {
                    tasksDB.showTask(results.rows.item(i), list);        #A
                }
                if(len === 0) {
                    var emptyItem = document.createElement("li");
                    emptyItem.innerHTML = '<div class="item_title">'
                      +'No tasks to display. <a href="#add">Add one</a>?';
                    document.getElementById("task_list")
                            .appendChild(emptyItem);
                }
           }, tasksDB.onerror);
        });
    }
},
showTask: function(task, list) {
    var newItem = document.createElement("li"), checked = '';
    var checked = (task.complete == 1) ? ' checked="checked"' : '';
    newItem.innerHTML = '<div class="item_complete">'                    #3
        +'<input type="checkbox" name="item_complete" id="chk_'          #3
        +task.id+'"'+checked+'></div>'                                   #3
        + '<div class="item_delete">'                                    #3
        + '<a class="lnk_delete" id="del_'+task.id+'">Delete</a></div>'  #3
        + '<div class="item_title">'+task.desc+'</div>'                  #3
        + '<div class="item_due">'+task.due+'</div>';                    #3
    list.appendChild(newItem);
    document.getElementById('del_'+task.id).onclick = function(e) {
        if(confirm("Are you sure wish to delete this task?",
          "Confirm delete")) {
            tasksDB.deleteTask(task.id);
```

```
        }
    }

    document.getElementById('chk_'+task.id).onchange = function(e) {
        var updatedTask = {
            id: task.id,
            desc: task.desc,
            due: task.due,
            complete: e.target.checked
        };
        tasksDB.updateTask(updatedTask);
    }
}
```

#1 Use a cursor to get all tasks
#2 Pass object to showTask
#A Pass object to showTask
#3 Attach event to new item

Although there is quite a bit of code here, it is relatively straightforward when you consider the work it is doing. The load function is responsible for retrieving the tasks from the IndexedDB or Web SQL database, iterating over the result set and passing each task to the showTask function, which will in turn render the task onto the "Task List" view in your application. The load function first uses the openCursor API #1 to retrieve all tasks from the database. It then traverses over each record, passing the task object to the showTask function #2 for rendering. It uses the cursor's continue API to move to the next record, and exits when there are no records left to render.

The showTask function accepts the task object and a handle to the task list element on the HTML page as arguments, constructs the list item and adds it to the list accordingly. If the task is marked as complete, this function will set the checkbox to be checked, and it then constructs the various parts of each list item, including the description, due date, checkbox and delete link. Next, the code gets a handle to the new delete link and attaches the deleteTask function to the click event for the link. Finally, the function gets a handle to the checkbox for the task in question and attaches the updateFunction to the checkbox's change event #3.

At this stage, the sample application should be fully functional. Try it out on your iOS or Android device, or indeed on any other desktop or mobile browser that has support for localStorage and either IndexedDB and/or Web SQL. If both IndexedDB and Web SQL are available, the application will favor the former by default. In the next and final section of this chapter, you will learn how to ensure that your application will work offline using an application cache manifest file. You should then have an entire application that stores all of its data on the client, and is usable both online and offline.

5.5 Going offline with an application cache manifest

Up until recently, Web applications have largely been used primarily in "connected" environments, on desktop or laptop computers where the majority of the time an Internet connection is available. However, as rich Internet applications become more prominent as realistic alternatives to their desktop counterparts, and as mobile applications continue to gather momentum, the need for Web applications to work in scenarios where connectivity is not available gets much stronger. You could be on a flight with no Internet available, on a subway or train where cell signal is too weak, or in a remote location where 3G is non-existent. Or, perhaps you need to use an application that typically only works inside a corporate Intranet. In each case, offline access is the answer.

HTML5 includes the ApplicationCache interface to enable Web applications to be used offline. Not only that, but it also allows you to improve the performance of your applications by designating that certain resources should be cached locally and reused where possible, improving response times and reducing server load. Once the application has been used online once, it can then be used offline after that, meaning it is usable in any of the previous scenarios. To take the example of a corporate Intranet, you could allow a user to work with a subset of features offline, storing the data into localStorage or the local IndexedDB on the client machine, and when the user is back inside the corporate network, synchronize this data back to the main central server-side database.

In this section, you will learn how to enable offline applications in HTML5 using the cache manifest file. You will discover the syntax and various sections you can use in this file to indicate which resources should be cached, which should not, and to define fallbacks for those scenarios where something that should be available is not for some reason. You will also learn how to correctly serve this file from your Web server, force resources to be downloaded again when they have been updated and listen for related events and detect the current status of the user (are they online or offline). Finally, you'll add a cache manifest for your sample application and test that it continues to work even in "Airplane mode" on a mobile device.

5.5.1 Introducing the cache manifest file

The application cache manifest, in its most basic form, is simply a list of Web resources that should be cached locally by the user agent and loaded from cache from that point on, unless the manifest itself has been updated. This enables the listed resources (and the HTML document that loads the manifest file) to be used offline. Take your "My Tasks" application, for example, which only has three files – index.html, style.css and app.js. The cache manifest file for this application would look as follows:

Listing 5.14. cache.manifest – Enabling "My Tasks" to be used offline

```
CACHE MANIFEST                                                    #1
index.html
style.css
app.js
```

#1 Always at top of manifest

In this example, the cache manifest simply includes the string CACHE MANIFEST on the first line #1, which must always be placed at the top of cache manifest files, then lists all of the resources that are to be cached, in this case the three files used by our sample application. In fact, you do not need to explicitly include any resources in the cache manifest file that will actually load the manifest itself, but the HTML5 specification recommends that developers include it anyway.

If you have a static website or a Web application that can run entirely on the client, the above manifest will be sufficient for your needs. It is far more likely, however, that you will have some pages or features that simply cannot be used offline. For these resources, you can simply define that they are not available offline at all, or you can provide a fallback which will be loaded in their place should they fail to load. To do this, you must define sections in your manifest file. There are three different sections you can use in a cache manifest:

- CACHE
  Any files listed under this section will be cached by the user agent when they are first downloaded, and re-used until the manifest has been updated. This is the default section, and does not need to be explicitly defined unless you are placing the section beneath another section in the manifest.
- NETWORK
  Any resources listed under this section will bypass the application cache, and will always be downloaded from the network. This means that these resources will never be usable offline. You can use wildcards to account for many resources at a time.
- FALLBACK
  This section allows you to defines a list of resource URIs along with fallback URIs that should be used in the event that the primary resource is not accessible. You must specify relative resource paths here, and they must be in the same origin (in other words domain and port number combination) as the manifest file itself. Again, you can use wildcards for the primary resource. The fallback URI must be specific, however.

Let's look at an example of a more complex cache manifest that implements these sections:

Listing 5.15. cache.manifest – Using manifest sections

```
CACHE MANIFEST
index.html                                                    #1
style.css
app.js

NETWORK:                                                      #2
*.jsp
/admin
/login
```

```
FALLBACK:                                                    #3
img/big.jpg img/small.jpg
*.html offline.html

CACHE:                                                       #4
help.html
```

#1 Don't explicitly define CACHE
#2 Online-only resources
#3 Defining fallback resources
#4 Multiple section definitions

The first thing you'll notice in listing 5.13 is that you do not explicitly define a CACHE section #1, and just list resources to be cached immediately after the opening line of the manifest. If you don't specify a section, it is assumed to be a CACHE section. You define a NETWORK section to list any resources that should never be cached or available offline #2. Dynamic scripts that rely on a server connection such as authentication scripts are a good example of a useful entry in this section. The FALLBACK section #3 defines some resources with fallback URIs that will be used in their place should they not be accessible to the application. Finally, you define a second CACHE section #4, but this time you must explicitly use the section heading as you have already defined other sections above it. In your sample application, you will use a NETWORK section with a wildcard entry, which will load the Google Web Font you are using from the network if it is available, otherwise it will simply not load it at all, and the app's main title heading will display in a standard Sans Serif font instead, such as Helvetica or Arial.

> **NOTE** The cache manifest will take precedence over any HTTP cache headers that may be used, and the restrictions on caching that typically impair secure Web pages served over Secure Sockets Layer (SSL) or Transport Layer Security (TLS) do not apply to the cache manifest, meaning that even secure Web applications can be used offline. In the next section, you will learn how to load the manifest in your HTML documents, and how to tell your Web server to provide the manifest file using the correct MIME type.

The order in which sections appear in the manifest does not matter, and as you can see from this example, you can define sections multiple times in the same manifest file.

5.5.2 Loading the manifest and serving up the correct MIME type

Telling an HTML document how to load a cache manifest is straightforward. You simply give your document's <html> element a manifest attribute with the path to the manifest file as the value. For example, if you name your manifest file cache.manifest, you would use the following code to load it in your HTML document:

```
<html manifest="cache.manifest">
  ...
```

```
</html>
```

In order for the manifest to be correctly loaded, however, your Web server needs to serve the file using the correct MIME type, `text/cache-manifest`. This is not typically set by default in a Web server's configuration file, so you will need to add it manually in order to use application caching. If you are using the Apache Web server, you can typically add MIME types by either modifying the httpd.conf configuration file or by serving a .htaccess file in the root of your Web application. In either case, to serve the correct MIME type for files with the extension .manifest, you need the following line:

```
AddType text/cache-manifest .manifest
```

If you are using the nginx Web server, you add MIME types by adding an entry to the mime.types file in the nginx conf directory. This file typically has the following format:

```
types {
  text/html                 html htm shtml;
  text/css                  css;
  text/xml                  xml;
  ...
}
```

To enable the cache manifest MIME type, add an entry into this file as follows:

```
  text/cache-manifest       manifest;
```

Restart your nginx server, and your cache manifest file should be served correctly from now on. If you are using another Web server, please consult your Web server's documentation for further information on how to add MIME types.

5.5.3 Forcing cache updates with revision numbers

As you have seen, the application cache manifest allows you to define resources that should be cached locally by the Web browser and available for use offline in the future. This is great, but what if you update your application or add additional functionality – how do you tell the browser to grab the latest version? If you're expecting it to fetch the resources any time the resources themselves change on the server, you're going to be disappointed. After all, in order to do this the browser would need to fetch the resources each and every time, and check if each resource is newer than its cached version. This negates the benefits of the cache manifest altogether as it slows down performance and adds unnecessary server load.

Instead, the server will only check to see if the cache manifest file itself has been updated. This means that we can force the browser to fetch the resources once again by modifying the manifest file. The best way to do this is using incremental revision numbers. For example, take the cache.manifest file you used in your "My Tasks" application: you can

include a comment in the file that contains a revision number, as shown in the following listing.

Listing 5.16. cache.manifest – Using revision numbers

```
CACHE MANIFEST
# Rev 1                                                    #A
index.html
style.css
app.js

NETWORK:
*                                                          #B
```

#A The pound symbol indicates a comment
#B Wildcard fallback to allow for Google Font API

If you subsequently update one of the files in this manifest and need to tell the browser to download the latest version, you simply increment the revision number to 2 and the next time the manifest is downloaded, the browser will see that it has changed and will re-download the resources into cache. You don't have to use revision numbers; changing anything at all in the manifest will cause the update. Revision numbers or timestamps that are updated using a build script probably make most sense, however, especially if you are building a large application.

The ApplicationCache object in HTML5 also contains an update API that allows you to programmatically update the cache. Again, the cache will only successfully update if the manifest file has changed. You can check that application caching is supported with the following code:

```
if('applicationCache' in window) {
    ...
}
```

When using the update API, you should check if an update is ready using the status property, and if it is you can then swap the new cache in for the old one as follows:

```
var c = window.applicationCache;
c.update();
if(c.status == c.UPDATEREADY) {
    c.swapCache();
}
```

One caveat with this approach is that it will force the cache itself to be updated, but it won't load the newly updated resources into the page. In order to do this, you would need to reload the page, which means between updating the cache programmatically and the page

reload, you are loading the new resources twice, which is typically unnecessary. In the next section, you will learn how to use application cache event handling to circumvent this issue.

5.5.4 Event handling and status detection

The HTML5 ApplicationCache API defines a series of events that allow you to monitor the state of the cache. Listening to these events can be useful for many things, particularly checking that the manifest file is being cached correctly and updated if required. Table 5.1 lists the various events available, with a brief description of each.

Table 5.1. Application Cache Events

Event	Description
`cached`	Fires when the manifest is cached for the first time
`checking`	Fires when checking for an update to the manifest
`downloading`	Fires when an update has been found and the user agent is downloading resources
`error`	Fires when an issue prevents the manifest from being downloaded
`noupdate`	Fires when the manifest has finished downloading for the first time
`obsolete`	Fires when the manifest is not available, removing the application cache altogether
`progress`	Fires when a resource in the manifest is being downloaded
`updateready`	Fires when the resources in the manifest have been updated

Let's look at an example of how to listen to one of these events in an application, checking if an update is available, if so swapping the cache accordingly and asking the user if they would like to reload the application. The code for this is relatively straightforward:

Listing 5.17. Allowing users to reload the page if a newer version of the application is available

```
window.onload = function() {
    var c = window.applicationCache;
    c.onupdateready = function(e) {
        if(c.status == c.UPDATEREADY) {
           c.swapCache();                                              #1
           if(confirm("A new version of this app is available. Reload?")) {
               window.location.reload();
            }
        }
    }
}
```

#1 Overwrite cache with new items

This code adds a function to the window load event that listens for the updateready event on the applicationCache object #1. When this event fires, you check that an update is indeed ready, and if so you swap the cache with the latest resources. You then tell the user that a new version of the application is available, and offer them the opportunity to reload the page there and then. Figure 5.5 shows this dialog in action after an update has taken place on the cache manifest file.

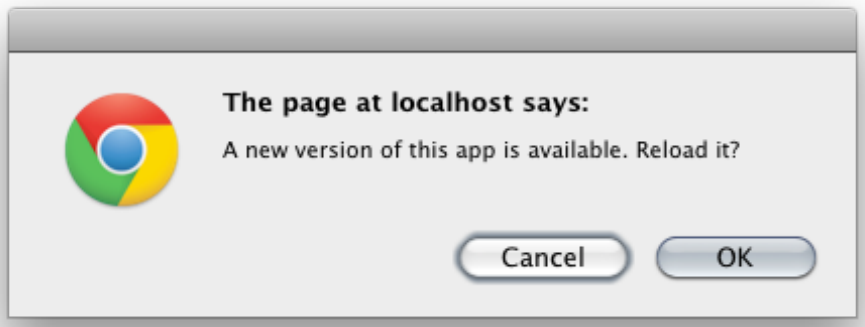

Figure 5.5. Listening to the updateready event. The code in Listing 5.17 will detect changes to files in the application, and will ask the user if they want to reload the app to ensure they are using the latest version.

Up until this point, all of the work on the cache manifest file has been manual. In the next section, you'll take all of what you've learned and apply it to the "My Tasks" sample application.

5.5.5 Adding offline support to the Tasks application

Fortunately, we've already covered everything you need to do to add offline support to the "My Tasks" application you've been building throughout this chapter. If you have been working with the application without a Web server, it's worth pointing out that you will not be able to use cache manifests unless your application resides on an actual Web server. You must then configure your Web server to serve the correct MIME type for cache manifest files, as described previously. Next, you should create the manifest file itself, which should contain the code in the following listing.

Listing 5.18. cache.manifest – "My Tasks" application cache manifest file

```
CACHE MANIFEST
# Rev 1
index.html
```

```
style.css
app.js

NETWORK:
*
```

Next, you need to modify the index.html file, and replacing the current <html> element line with the following:

```
<html lang="en" class="blue" manifest="cache.manifest">
```

If you have followed these steps correctly, you should now be able to use your application offline. One way to verify this is to use a mobile device to access the application. Make sure you're able to connect to your Web server first, and load the application for the first time. Now, turn on "Airplane Mode" on your device, which should kill all network connectivity. Refresh the page in your device's Web browser, and you should still be able to use the application in full. The result can be seen in the screenshots in figure 5.6.

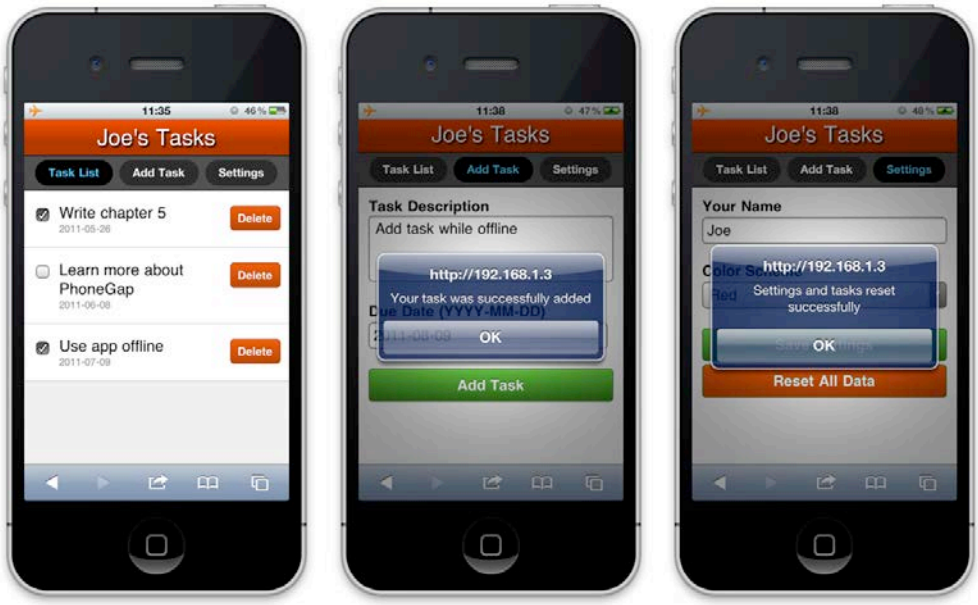

Figure 5.6. "My Tasks" application running offline. You'll notice the airplane icon in the top right indicating that the phone has no network access. You may also notice that the jazzy font we used in the heading is no longer showing – this font was loaded from the Google Font API, which isn't available when we're offline.

Note that these screenshots were taken from a physical iPhone 4, and that the iOS Simulator for Mac OS X does not feature an "Airplane Mode" setting like the physical device does.

5.6 Summary

HTML5 is finally providing the Web with a solution to the problem of working offline with client-side data storage and application-based caching controls. In the past there have been a plethora of solutions for saving Web pages for later use in an offline environment, but until now there has been no solution for using Web applications in such a manner. By allowing Web apps to store data locally on the client, HTML5 enables these applications to work without needing to be connected to a central server all of the time. Using cache manifests, you can now finally tell Web browsers what parts of the application it should cache, and just as important, what parts it shouldn't. You can even define fallbacks that will be used should the network be unavailable.

With the techniques outlined in this chapter, you can begin to build offline-capable applications of your own. An example where client-side data storage and application caching would be extremely useful is a mobile user interface for a Customer Relationship Management (CRM) application, where a member of the sales team can use the application on the road, any time, anywhere. If they are entering an area with poor network coverage, such as a remote location or an underground train, they can still use the application with any of the data that has already been downloaded to the device, and they can enter new data, which will be stored temporarily on the device itself, and synchronized back to the central server when the network is available again. All of this would previously have been impossible with a Web application, but thanks to HTML5 and advances in mobile phone technology, it is usable right now.

Another example of where client-side data storage is extremely useful is in the context of an HTML5 game (like the ones you will build later in this book). Rather than storing game saves and state data on the server, you can increase performance and reduce the latency and load by saving this data locally instead. Add a cache manifest to the equation and you have a game that can be run completely offline.

In the next chapter, you will learn about the 2D canvas API and how it allows you to build animations and games using native JavaScript APIs. You'll even get to build an entire game, complete with collision detection and keyboard controls!

6

2D Canvas

This chapter covers

- What is Canvas?
- Creating shapes, paths, and text
- Creating animation
- Simple physics
- Creating an HTML5 game from scratch

For many years developers were forced to use Adobe's Flash when creating highly interactive web applications. Sadly, Flash was left in the cold when it came to the mobile market explosion. Dark days without an alternative have ended, due to the birth of HTML5's Canvas API. Its scope is quite simple, but Canvas is capable of emulating medical training procedures, creating an interactive lobbying presentation, and even building supplementary education apps.

Throughout this chapter, you'll explore the 2D Canvas API's ability to processes data and how to set up the necessary HTML. After that you'll create and animate unique shapes with JavaScript and Canvas. These principles could be carried over into creating a full length animation, graphing data onto a rotating globe, or a drawing application. Instead, you'll use them for the true reason for all technology: Creating games!

You'll create the game of Breakout, which includes animated elements (a ball and paddle), collision detection, and keyboard/mouse controls. Upon putting all the components together, you'll take everything a step further into a fully polished product, which includes a score counter, progressively increasing difficulty, and an opening/closing screen. Adding polish greatly helps to monetize a game's worth, resulting in a better return on investment (ROI). After completing this chapter on 2D Canvas, you'll have learned all the necessary tools to build your own Canvas applications from scratch.

6.1 What is the Canvas context?

Before working with Canvas you must set up drawing tools (also known as setting the context). The most commonly used context draws a two dimensional plane (2D) where everything is flat. With 2D, games like Super Mario, drawing applications similar to Photoshop, or small animated films can be created. The second context is a set of three dimensional (3D) drawing tools. While 3D context allows for more robust graphics such as models from Toy Story and first person shooters, it's not as stable across browsers (more about 3D in chapter 9).

Canvas: A product of Apple's IOS

Canvas wasn't dreamed up overnight by the W3C for HTML5. It originally came in 2004 as part of the Mac OS X Webkit by Apple. 2 years later Gecko and Opera browsers adopted it. As popularity grew, Canvas would eventually be accepted as an HTML5 standard.

Because 2D is great for programming simple games, we'll guide you through building Breakout with the 2D context by using JavaScript and HTML. As you'll soon see a majority of the creation process involves accessing the 2D Context API via JavaScript, so you can send an object to the CanvasRenderingContext2D interface object. While "CanvasRenderingContext2D interface object" sounds long and fancy, it really means accessing Canvas for object creation. After that you create gradients, lines, shapes, and so on. Each creation is drawn on top of any previously created objects. To help better understand creating shapes, we've simplified the process into four simple steps:

1. 2D Context is accessed with JavaScript.
2. JavaScript requests an object from the API (gradient, square, line, etc).
3. Canvas draws the latest object on top of previous elements.
4. Rinse and repeat for each new object.

Each object is layered on a simple graph system inside the <canvas> tag as shown in figure 6.1. At first glance the graph appears to be a normal Cartesian graph. Upon further investigation you'll notice the starting point is located in the top left corner. Another difference is the y axis increases while moving downward, instead of incrementing upwards. Now that you understand how Canvas creates and positions elements, let's start practicing with 2D shapes by setting up the core HTML and JavaScript.

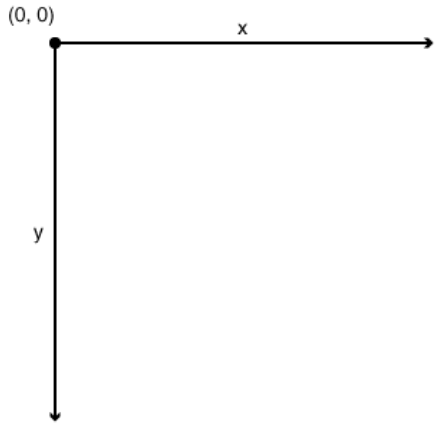

Figure 6.1 The invisible Cartesian graph all Canvas objects are created on. Notice that the x and y coordinates begin in the top left and the y axis increments downward.

6.1.1 Preparing the Canvas

To start preparing Canvas for the game open a text editor to create a document called index.html. Inside the file place the <canvas> tag into the <body> with an id, width, and height attribute. If these properties aren't declared the browser will use default sizes and make it difficult to control the viewing window. In addition, you can add a short and sweet message for browsers who can't support the <canvas> tag.

Listing 6.1 index.html – Default Canvas HTML

```
<!DOCTYPE html>

<html>
<head>
    <title>Canvas Demo</title>
</head>

<body>
    <canvas id="canvas" width="408" height="250">
        Your browser shall not pass! Download Firefox to view this content.
#A
    </canvas>
</body>

</html>
```

#A Replaced when Canvas loads

After refreshing your browser to see the almighty <canvas> element, it seems to be invisible or broken. To get the page working create a JavaScript file named shapes.js and fill it with the following snippet to access the Canvas API.

```
var canvas = document.getElementById('canvas');
```

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

The previous chunk of JavaScript won't work until you've loaded it into the HTML document. Inject it into the page through a quick snippet of code at the bottom of the canvas.html file right before the </body> tag.

```
<body>
    <canvas id="canvas" width="408" height="250">
        Your browser shall not pass! Download Firefox to view this content.
    </canvas>
    <script type="text/javascript" src="shapes.js"></script>
</body>
```

Canvas's context element is not only useful for defining 2D drawing; you can also use it for browser sniffing. Simply encapsulate the context variable in an if statement and you won't have to worry about unsupportive browsers gagging to death on the JavaScript file.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');
}
```

That's all you need to do to set up the drawing surface with browser fallbacks. Now populate your drawing area with objects created through the Canvas API. Before you start drawing elements and animating them, you need to know how data is processed. This is important because drawing objects is performed in a manner most developers haven't encountered.

Supporting Internet Explorer

With any new web technology, Internet Explorer (IE) brings its personal issues into your life. Versions 7 and 8 will refuse API commands unless you use explorecanvas (http://code.google.com/p/explorercanvas). IE 9 gives decent support, while IE 10 is looking quite solid. Our disclaimer for explorecanvas is that you can do simple animations with it, but more advanced support (our Breakout game tutorial) isn't guaranteed.

6.1.2 Canvas data processing

When creating Canvas objects through JavaScript, the browser recreates all graphical assets from scratch. Everything must be recreated due to the Canvas' graphic technology called bitmap (which has a memory span shorter than a goldfish). Bitmap creates graphics by storing graphical data in an organized array of data. When processed by a computer it spits out pixels to create an image. Due to a bitmap processing method, the drawing surface must be wiped clean and repopulated with a fresh set of data continuously. If it isn't, the new graphics will overlap the previous. Failure to prevent overlapping creates a huge mess of ambiguous shapes. Knowing these data processing limitations, you should be able to spot any odd or buggy behaviors that may occur.

SVG + CANVAS

If you're wondering why Canvas is designed to infinitely recreate images, then you aren't alone. Many people have asked "Why did Apple use a bitmap based system for Canvas?" when a solution exists that doesn't require everything to be constantly redrawn (Scalable

Vector Graphics aka SVG). On the other hand, Canvas is currently stomping SVG in popularity. One could explain Canvas' triumph through the lack of awareness and knowledge developers have of SVG. Now that you've learned how to prepare your drawing surface and create objects, you're ready to dive into drawing.

6.2 Drawing objects

Objects create artwork, animations, and most importantly your HTML5 game. These shapes are usually circles or squares filled with a variety of properties such as color, gradients, and/or images. Canvas libraries like PaperJS combine these elements to create interactive and engaging content seen in figure 6.2. While the colorful example looks fascinating, it takes a lot of code and cleverly thought-out logic to create.

Figure 6.2 PaperJS is a Canvas library focused on making objects interactive and animated. The creators feature a demo on their website where the rainbow follows and dynamically changes according to your mouse movements. You can interact with the live demo at http://paperjs.org/examples/nyan-rainbow/

When creating complex characters or backgrounds on the drawing surface, you'll need to think creatively. Crafting graphics usually involves breaking shapes into simpler pieces such as circles and/or squares. If you can't break a core composition into elements, then you'll have to set up a path. Paths use the points they receive to create complex shapes such as the body of a car. An alternative to shapes and paths is pulling an image into Canvas. While directly embedded images have the least flexibility, they can also save valuable development time. To help learn how to create your game's graphics, we will walk you through using

simple shapes, paths, images, and text. After assembling a few Canvas demos, we'll demystify animation with a couple lines of code capable of animating your Breakout game.

> **What! No box model?**
>
> If you've worked with CSS in a browser, than you're probably used to working with the box model. Canvas doesn't make use of it, meaning objects will not grow and shrink to the proportion of their container, instead they overflow without being checked. For example, a line of text that's too long won't automatically wrap to fit the <canvas> tag's width and height. Just like the box model isn't there, CSS is also missing. The only miniscule amount you'll see of it is the ability to add some minor styling through the API.
>
> While working on your graphics in the proceeding pages you'll notice the box model's absence is a great thing. Use overflow to your advantage by making objects appear to vanish off the page. As for CSS, you're out of luck and its absence can really make typography a struggle.

6.2.1 Shapes, paths, images, and text

Have you ever created artwork in Photoshop or Gimp? By creating objects in Canvas you're almost doing the exact same thing. The major difference is the lack of a user interface, so all the shapes must be created through code. You can create a simple shape such as a square, complex shapes through paths, embed images, and add text.

> **HTML5 Cheat Sheet**
>
> We highly recommended downloading and printing the HTML5 Cheat Sheet for reference from nihilogic at blog.nihilogic.dk/2009/02/html5-canvas-cheat-sheet.html

SHAPES

The easiest object to create is simple shapes (rectangles). They're drawn as clear, filled, or outlined and accept four parameters as seen in figure 6.3. The first two parameters determine the spawning position on the graph (x and y). Although the current viewing space only shows positive x and y coordinates, you may spawn shapes at negative values. The next two parameters specify the width and height in pixels.

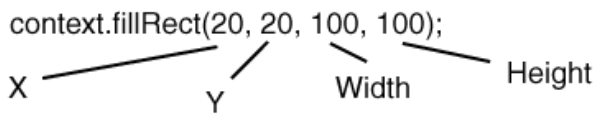

Figure 6.3 To create a rectangle you'll need to give four different parameters. The current figure would create a 100 x 100 pixel square at the 20 pixel x and y position. Currently rectangles are the only basic drawing component in Canvas. To create more complex items you'll need to use shapes.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    context.fillRect(20,20,100,100);
}
```

While a square really isn't much to look at, a few more shapes could create Wilson from figure 6.4, who is the friend of all developers locked away in their cubicles. To get a nicer looking square add two rectangles for eyes and a transparent rectangle to erase part of the square for a mouth. Change the square's fill color from the default black to a nice shade of yellow. Order in listing 6.2 is of the utmost importance, as Canvas draws each new item on top of the previous.

Figure 6.4 Say hello to Wilson, friend to all developers stranded in their cubicles. He is composed of three shapes on top of a large yellow square. For the eyes you'll add rectangles while the mouth is created by erasing part of the larger square.

Listing 6.2 shapes.js – Layering shapes

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    context.fillStyle = 'yellow';                                    #A
    context.fillRect(20,20,100,100);
    context.fillStyle = 'grey';
    context.fillRect(40,50,10,10);
    context.fillRect(90,50,10,10);

    context.clearRect(30,100,80,10);                                 #B
}
```

#A Defaults to black unless specified
#B clearRect erases all elements under it

Your new friend doesn't look the greatest... How about making him more stylish with a pair of black-rimmed glasses, as seen in figure 6.5? To accessorize him, use .strokeRect to draw

simple boxes around the eyes. Add a value of 3 pixels to .lineWidth for a thicker black line.

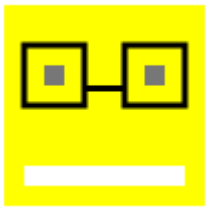

Figure 6.5 Wilson's trendy developer glasses were created through square shapes. The major difference is we filled in the outline of the square instead of filling it.

Add the following code after .clearRect and you'll be good to go.

```
context.fillStyle = 'black';
context.lineWidth = 3;
context.strokeRect(30,40,30,30);
context.strokeRect(80,40,30,30);
context.fillRect(60,60,20,3);
```

Not bad, you've created and drawn your first piece of artwork in Canvas. While it's a rather handsome, stereotypical smiley face, it feels... a bit blocky. You can smooth it out through using paths instead of simple shapes.

PATHS

One or more segments compose a path to create circles, trapezoids, and other complex shapes. To start drawing a complex shape tell the API to enter path drawing mode via beginPath(). After that declare a starting point with moveTo(x,y) if necessary. Then designate an x and y coordinate to draw with lineTo*(x,y)*. Lastly, create the line with a *stroke()*. The standard order you'll need to remember is beginPath(), moveTo(), lineTo(), and stroke(). Note that you can should use multiple moveTo() and lineTo()'s to create complex shapes. Here's the step-wise process:

1. Start drawing a path with .beginPath().
2. Use moveTo(x,y) to move the path without drawing on the Canvas (optional).
3. Draw lines as needed with lineTo(x,y).
4. Use items 2 and 3 as often as you want.
5. Complete the path by using stroke().

Joints of connecting path lines may be adjusted through lineJoin to round (slightly rounded edges), bevel (breaks off connecting line's tip), or miter (the default). Lines can also be adjusted through the lineCap (where the line tip ends) to round for a less jagged end point. While lineJoin and lineCap create circular lines, you'll need arcs to make real circles. Create

an arc with arc(*x*, *y*, *radius*, *startAngle*, *endAngle* [, *anticlockwise*]), as illustrated in figure 6.6.

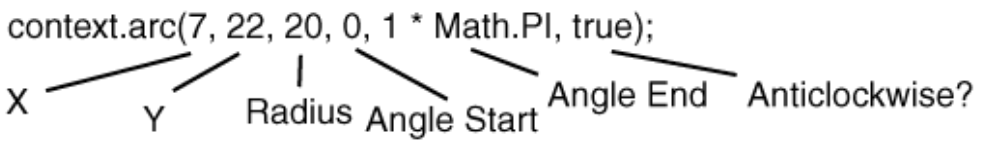

Figure 6.6 This arc will create a shape at 7, 22 (x, y) on the graph. Because the angle starts from 0 and goes to 1 pi, it will create a half circle. It should be noted that 2 pi is equal to a whole circle. Lastly, declaring true will set the shape to be created in a reverse clockwise manner.

Use context.arc() to create a circular shape then manipulate it with styling. Note that the starting point is in the center because an arc is drawn from the center outwards. Give the arc() a radius in pixels, a startAngle, and endAngle which actually creates the circle. StartAngle is usually best kept at 0∏, while the endAngle is 2∏ since it's the complete circumference of a circle (using only ∏ will create half a circle). Lastly, setting anticlockwise to true or false will determine whether your angle is drawn clockwise or counterclockwise. Anticlockwise is usually used to flip a half circle upside down.

Where is pi on my keyboard?

To prevent you from making my mistake of searching for pi on the keyboard, we'll tell you now that it isn't there. Instead use the built-in JavaScript math function for Math.PI and multiply it by 1 or 2.

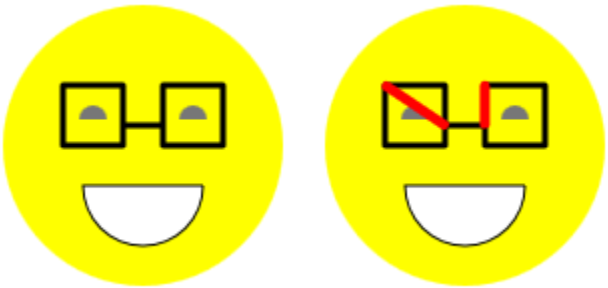

Figure 6.7 Creating a happier Wilson with paths. Red on right demonstrates using moveTo() to create lines without fill.

To make Wilson a bit more appealing specify information to smooth out his glasses, create a more circular face, and half-circles for the eyes. Listing 6.3 has all the information you need

to create a nicely rounded smiley face. Make sure to look at figure 6.7 to see how moveTo(x,y) commands were used to create Wilson's glasses.

Listing 6.3 paths.js – Mastering paths

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    context.beginPath();
    context.arc(70,70, 70, 0, 2 * Math.PI, false);                    #A
    context.fillStyle = 'yellow';
    context.fill();

    context.beginPath();
    context.arc(45, 57, 7, 0, 1 * Math.PI, true);
    context.moveTo(100,57);                                           #B
    context.arc(95,57, 7, 0, 1 * Math.PI, true);
    context.fillStyle = '#777777';                                    #C
    context.fill();

    context.beginPath();
    context.arc(70,90, 30, 0, 1 * Math.PI, false);
    context.lineTo(100,90);
    context.fillStyle = '#ffffff';
    context.fill();
    context.stroke();

    context.fillStyle = 'black';
    context.lineWidth = 3;
    context.lineJoin = 'round';
    context.lineCap = 'round';
    context.beginPath();
    context.moveTo(30,40);
    context.lineTo(30,70);
    context.lineTo(60,70);
    context.lineTo(60,40);
    context.lineTo(30,40);
    context.moveTo(60,60);
    context.lineTo(80,60);
    context.moveTo(80,40);
    context.lineTo(80,70);
    context.lineTo(110,70);
    context.lineTo(110,40);
    context.lineTo(80,40);
    context.stroke();
}
```

#A False draws arcs clockwise
#B MoveTo() doesn't create visible lines
#C Hex values okay for color

IMAGES

When using an image in Canvas, it's not as simple as injecting an <img> tag in HTML. You're required to create an image variable, store its location, and then draw it. The process is quite

robust for adding a simple picture, but Canvas requires all elements to be imported as objects. Think of it as prepping the data so Canvas can properly swallow it without choking.

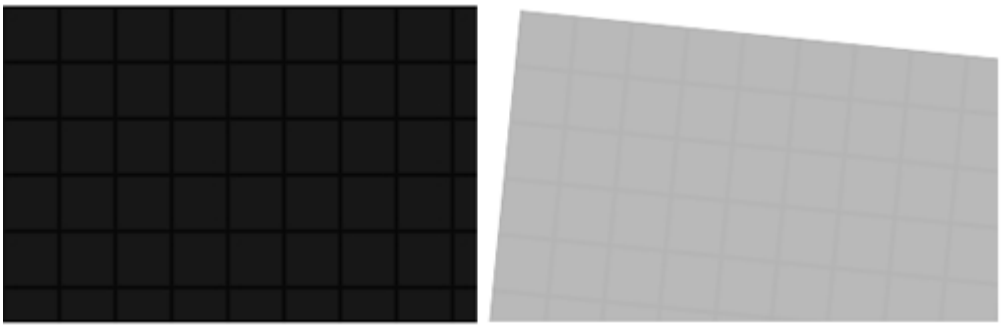

Figure 6.8 Properly loaded and displayed image on left. On the right an opacity and rotate style is being applied. Note that rotate tilts an object from the center.

Insert a background image by grabbing background.jpg off of manning.com and using the following code snippet to create figure 6.8's left box.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    var img = new Image();
    img.src = 'background.jpg';                                    #1
    context.drawImage(img,10,20);
}
```

Notice the image #1 is larger than the drawing surface. Overflowing images won't cause problems and perfectly acceptable. You can also insert images smaller than the drawing area.

Styling is always added to images and shapes before creating them. Properties such as angle, opacity, shadows, and more can be added. Recreate the right side image in figure 6.8 by rotating and changing the opacity of your current image. To alter these properties set the rotate and globalAlpha to a value between 0 and 1.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    var img = new Image();
    img.src = 'background.jpg';
    context.rotate(0.1);              #A
    context.globalAlpha = 0.3;
    context.drawImage(img,30,20);
}
```

#A Calculated in radians

TEXT

The Canvas 2D context API gives developers a decent amount of control over text styling. Just like CSS you have access to text align, vertical-align (called text baseline), and @font-face fonts. On the other hand, some of the other CSS properties can't be tweaked, such as letter spacing.

Warning: Avoid bitmap fonts

> Make sure to use vector based fonts instead of bitmap because "transformations would likely make the font look very ugly" (W3C Canvas Working Draft). Using a bitmap-based font will corrode your text in a macabre display of typography when rotated.

While you've probably heard of text align, you might not have heard of text baseline. Through the function .textBaseline you can get more flexibility where the bottom of letters line up. While .textBaseline is usually for other languages, use it for aligning text on the Cartesian Graph in Canvas. For instance, if you set .textBaseline to bottom, the x and y coordinate would place the text at the bottom of the letters instead of at the default top. Let's create the text in figure 6.9 with what we've learned here.

Figure 6.9 Filled vs. outlined text. Sadly no letter spacing option is available.

First, you define the font size and family properties. After that draw the text via fillText() and strokeText(), followed by x and y coordinates. You could also define a fourth optional field for the text's maximum width in pixels before wrapping. If you don't want sentences to break because they're longer than their container, declare the fourth field.

```
var canvas = document.getElementById('canvas');

if (canvas.getContext){
    var context = canvas.getContext('2d');

    context.font = '20px impact, helvetica, arial';             #A
    context.textBaseline = 'middle';
    context.fillText('I <3 Canvas', 30, 30);
    context.strokeText('I <3 Canvas', 60, 60);
}
```

#A Accepts parameters just like the font CSS declaration

After using the previous code snippet, you'll probably notice some adverse affects, easily seen in figure 6.9. Since you can't control letter spacing for strokeText(), it makes the words

look cramped. While ugly text might not be a big deal for some, typophiles aren't going to be happy until Canvas gives more control over typography.

6.2.2 Animating with Canvas

When animating objects with Canvas, you might be deceived into thinking it performs animation. Well, it does - but it doesn't. Truth is, Canvas relies on JavaScript intervals to draw frames. Using an interval in unison with drawing shapes creates the illusion of movement. Additional equations can then be added for gravity and collisions.

To create moving objects declare how often your interval is drawn in milliseconds (1,000 milliseconds = 1 second) and point it to a function with animation logic. When the countdown is complete, the function will be fired. After that, the interval continues the process of counting down and firing until it's been shutdown.

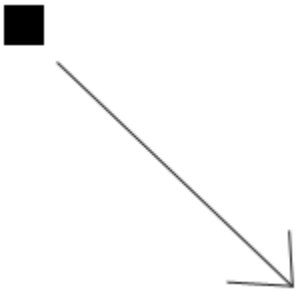

Figure 6.10 Through using an interval the square will move from the top left to the bottom right. The square will keep on moving past the Canvas edge until it flies off the page.

With Listing 6.4 use setInterval(function, interval) to create an animation like figure 6.10. Using the interval update time from the listing should be fine, but performing extremely complex equations will require a larger number. Usually you don't have to worry, as modern day computers can handle a couple complex intervals running at the same time. In a case where the animation needs to be killed in order to free up memory, call clearInterval().

More about intervals

If you've never worked with intervals before, we highly recommend learning more after completing your Canvas game. They are useful for a variety of front-end needs (sliders, auto updating applications, and tracking software just to name a few). For more in-depth information see Matt Doyle's article titled JavaScript Timers with setTimeout and setInterval at http://www.elated.com/articles/javascript-timers-with-settimeout-and-setinterval/.

Listing 6.4 animation.js – Simple animation

```
var canvas = document.getElementById('canvas');
var context;
var x = 10;
var y = 5;
var xAdd = 2;                                    #A
var yAdd = 3;                                    #A

function init() {
    if (canvas.getContext){
            context = canvas.getContext('2d');
            return setInterval(draw, 12);   #B
    }
}

function draw() {
    context.fillStyle = 'black';                 #1
    context.fillRect(x,y,20,20);
    x += xAdd;
    y += yAdd;
}

init();
```

#A Determines animations speed

After refreshing your browser and running the previous chunk of code, you'll notice a HUGE problem. Canvas is updating with the interval, but dumping every animation frame onto the page. Fix the messy output by drawing a clear rectangle at the beginning #1 of the draw function.

```
function draw() {
    context.clearRect(0,0,408,250);
    context.fillStyle = 'black';
    context.fillRect(x,y,20,20);
    x += xAdd;
    y += yAdd;
}
```

Canvas clearing alternatives

Some people have realized that setting a width will reset the Canvas frames. While changing the width sounds more clever than drawing clear rectangles, its known to cause instability in some browsers. We recommend using clear rectangles to erase all the previously drawn frames instead of fiddling with the width constantly.

A SETINTERVAL ALTERNATIVE

setInterval()'s original design is meant for light equations and not robust formulas found in video games. In response browser vendors have created a JavaScript function known as requestAnimationFrame() that will interpret the appropriate number of frames to display for a user's computer. The bad news is the feature isn't supported in all browsers. To find out

more see the article at Mozilla's Developer Network called requestAnimationFrame (https://developer.mozilla.org/en/DOM/window.requestAnimationFrame).

With a thorough understanding of how drawings are combined with animation you have all the necessary tools to create your own game. For the next step we'll walk you through creating Breakout's assets, organizing game data, and how to set up your default JavaScript structure.

6.3 Creating a Canvas game

In the past you'd develop browser games like Breakout, shown in figure 6.11, through Flash. While Flash runs across all browsers, support is terrible on mobile devices, plus it requires a plugin. Now, with Canvas 2D Context you have all the benefits of Flash, and your games will run on mobile devices and without a plugin. In addition one HTML5 Canvas application can be distributed across multiple platforms through mobile frameworks like PhoneGap.com.

Your first Canvas game will make use of physics, advanced animation, keyboard/mouse controls, and polishing. While physics and advanced animation might sound scary, no prior knowledge is necessary and we'll walk you through each step of the way.

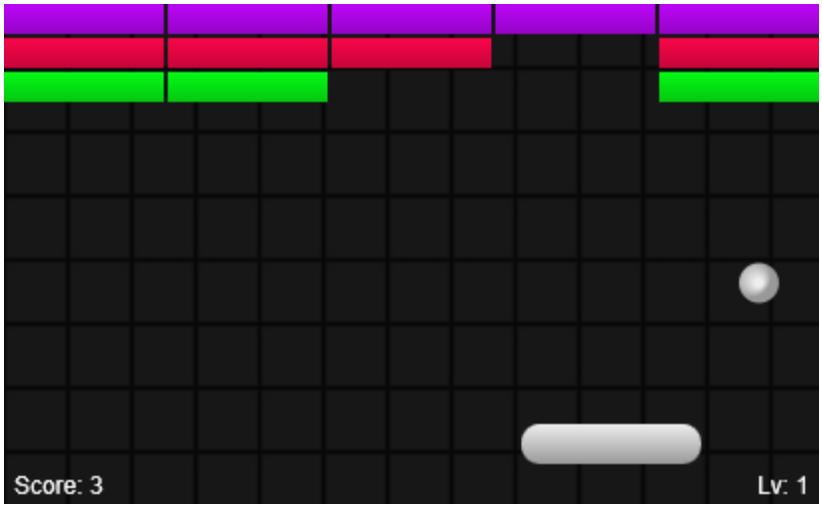

Figure 6.11 Breakout's objective is to bounce a ball via paddle to break bricks. When the ball goes out of bounds the game shuts down. You can play the game now at html5inaction.com and download all the files needed to complete your own Breakout game from manning.com.

Before you create Breakout, download the book's complimentary files from manning.com. From html5inaction.com you can test drive the game and see all the cool features in action. For the rest of this section you'll inflate a ball, carve a paddle, lay down the bricks, and then

bring it all together. But first, you need to lay down the foundation - setting up your file and Canvas. Let's get started!

6.3.1 The foundation

Set up the game's HTML in a file called breakout.html and make the <canvas> tag 408 by 250 pixels so there's lots of room to work with. After that add an inline style to center everything

```
<body style="text-align: center">
    <canvas id="canvas" width="408" height="250">
        Download FireFox to play this game now!
    </canvas>

    <script type="text/javascript" src="game.js"></script>
</body>
```

Next, create and point to a new JavaScript file called game.js (show in listing 6.5). With a solid HTML setup, you can use the DOM to collect default variables to speed up game development. You'll find these variables useful when retrieving the <canvas> width and height for calculating equations. In addition, lay down a simple background image that takes up the whole play area.

Listing 6.5 game.js – Default JavaScript

```
var canvas = document.getElementById('canvas');
var canvasAtt = canvas.attributes;
var canvasW = canvasAtt.getNamedItem('width').value;
var canvasH = canvasAtt.getNamedItem('height').value;
var canvasRun;

if (canvas.getContext){
        var context = canvas.getContext('2d');
        background();
}

function background() {
        var img = new Image();
        img.src = 'background.jpg';
        context.drawImage(img,0,0);
}
```

6.3.2 Inflating a ball

To create the ball in listing 6.6 use the same principles you learned to create Wilson the smiley face. Instead of leaving the ball's background plain, make use of a gradient to add some interest. The catch is you have to dynamically generate the gradient at the ball's position. Otherwise the gradient won't follow the object when animated.

Listing 6.6 game.js – Ball creation

```
var ballX;                                                    #A
var ballY;                                                    #A
var ballR;
```

```
if (canvas.getContext){
    var context = canvas.getContext('2d');
    background();

    ballInit();
    ballDraw();
}

function ballInit() {                                          #B
    ballX = 120;
    ballY = 120;
    ballR = 10;
}

function ballDraw() {                                          #C
    context.beginPath();
    context.arc(ballX, ballY, ballR, 0, 2 * Math.PI, false);
    context.fillStyle = ballGrad();
    context.fill();
}

function ballGrad() {
    var ballG = context.createRadialGradient(ballX,ballY,2,ballX-4,ballY-
3,10);
    ballG.addColorStop(0, '#eee');
    ballG.addColorStop(1, '#999');
    return ballG;
}
```

#A Store ball's position here
#B Initial setup
#C Animate the object

6.3.3 Carving a paddle

To create a paddle follow listing 6.7 to combine four arcs into a pill shape. Problem is you can't use the arc from the smiley face exercise. Instead use arcTo(x1,y1,x2,y2,radius), which allows circular shapes to be created as part of a path for an object. In addition add a gradient like the ball, except add a linear version with createLinearGradient(x1,y1,x2,y2).

Listing 6.7 game.js – Paddle creation

```
var padX;
var padY;
var padW;
var padH;
var padR;

if (canvas.getContext){
    var context = canvas.getContext('2d');
    background();

    ballInit();
    ballDraw();
```

```
    padInit();
    padDraw();
}

function padInit() {
    padX = 100;
    padY = 210;
    padW = 90;
    padH = 20;
    padR = 9;
}

function padDraw() {
    context.beginPath();
    context.moveTo(padX,padY);                                          #A
    context.arcTo(padX+padW, padY, padX+padW, padY+padR, padR);
    context.arcTo(padX+padW, padY+padH,padX+padW-padR,padY+padH, padR);
    context.arcTo(padX, padY+padH, padX, padY+padH-padR, padR);
    context.arcTo(padX, padY, padX+padR, padY, padR);
    context.closePath();                                                #B
    context.fillStyle = padGrad();
    context.fill();
}

function padGrad() {
    var padG = context.createLinearGradient(padX,padY,padX,padY+20);
    padG.addColorStop(0, '#eee');
    padG.addColorStop(1, '#999');
    return padG;
}
```

#A Sets the paddle's spawn origin
#B Closing paths can prevent buggy behavior

6.3.4 Laying down bricks

To get the width for each brick you'll need to do some calculations. Since you have 5 bricks with a total of 4 gaps that need spacing. Place a 2px gap between each brick to take up a total of 8px (4 gaps x 2px). Remove the 8px from the 408px Canvas width for a total length of 400px. Since 5 bricks need to fit inside the remaining width, you're looking at 80px per brick (400px / 5 bricks = 80px). Want to use 9 bricks instead? You'll have to remove the 8 2px gaps from the width, then adjust the <canvas> width to a number that's perfectly divisible by 9 bricks. If you don't adjust the width, you'll end up with a decimal that offsets your Breakout game. Having trouble following our logic? Look at the diagram in figure 6.12 for a more visual explanation.

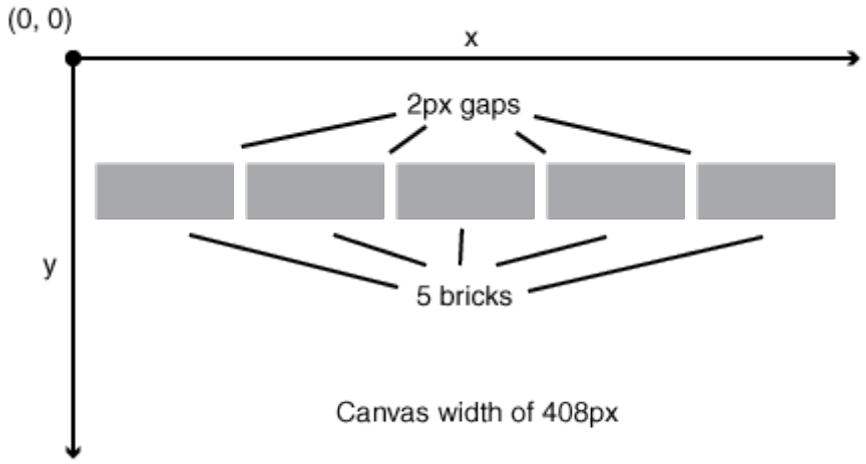

Figure 6.12 With a total of 4 gaps at 2px each. The totaling 8px needs to be subtracted from the <canvas> width leaving 400px. Distribute the remaining width to each brick, leaving 80px for each (400px / 5 bricks = 80px).

Bricks could be placed by rewriting a basic shape command over, and over, and over. Instead, create a two dimensional array as seen in listing 6.8 to hold each brick's row and column. To lay down the bricks, loop through the array data and place each according to its row and column.

Listing 6.8 game.js –Brick array creation

```
var bricks;
var bGap;
var bRow;
var bCol;
var bW;
var bH;

if (canvas.getContext){
    var context = canvas.getContext('2d');
    background();

    brickInit();
    brickDraw();

    ballInit();
    ballDraw();

    padInit();
    padDraw();
}
```

```
function brickInit() {
    bGap = 2;
    bRow = 3;
    bCol = 5;
    bW = 80;
    bH = 15;

    bricks = new Array(bRow);
    for (i=0; i<bRow; i++) {                                    #A
        bricks[i] = new Array(bCol);                            #A
    }                                                           #A
}

function brickDraw() {
    for (i=0; i<bRow; i++) {                                    #B
        for (j=0; j<bCol; j++) {                                #B
            if (bricks[i][j] !== false) {                       #B
                context.fillStyle = 'blue';
                context.fillRect(brickX(j),brickY(i),bW,bH);    #1
            }
        }
    }
}
```

#A Array creation loop
#B Array execution loop

While outputting the bricks, you need to be specific about what x and y coordinates they're placed at. Add two functions #1 to handle the column and row setup with the following code snippet.

```
function brickX(row) {
        return (row * bW) + (row * bGap);
}

function brickY(col) {
        return (col * bH) + (col * bGap);
}
```

Since the brick placement is taken care of, why don't you make them a little more colorful? Change up the graphics by coloring each brick based upon its row via a switch statement. In order to make listing 6.9 function correctly replace *context.fillStyle = 'blue';* with *context.fillStyle = brickColor(i);*, otherwise the color won't change.

Listing 6.9 game.js - Coloring bricks

```
function brickColor(row) {
    y = brickY(row);
    var brickG = context.createLinearGradient(0,y,0,y+bH);
    switch(row) {
        case 0:  brickG.addColorStop(0,'#bd06f9');              #A
        brickG.addColorStop(1,'#9604c7'); break;                #A

        case 1: brickG.addColorStop(0,'#F9064A');               #B
        brickG.addColorStop(1,'#c7043b'); break;                #B

        case 2: brickG.addColorStop(0,'#05fa15');               #C
        brickG.addColorStop(1,'#04c711'); break;                #C
```

```
            default: brickG.addColorStop(0,'#faa105');                    #D
            brickG.addColorStop(1,'#c77f04'); break;                      #D
        }
        return context.fillStyle = brickG;
    }
```

#A Row 1 red
#B Row 2 purple
#C Row 3 green
#D Row 4+ orange

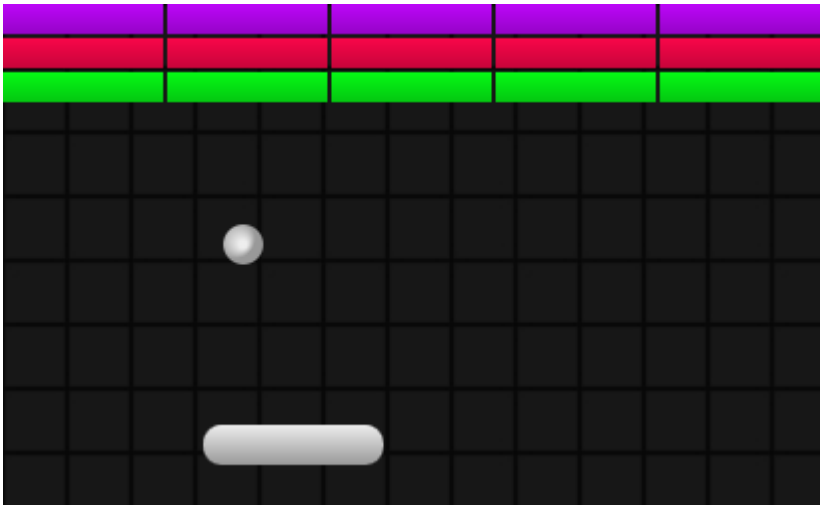

Figure 6.13 Using the previous code snippets you created a ball, paddle, and bricks with gradients. After refreshing your browser the current result should look like this.

6.3.5 Assembling a static game

Since you just wrote a lot of code make sure the draw() and init() functions look identical to listing 6.10. Your JavaScript file should be structured so variable declarations reside at the top, all object initialization function placed inside init(), and all draw functions placed inside draw(). Doing so will properly prepare objects to be animated and interacted with in the next section.

Before proceeding, make sure that your current progress is identical to figure 6.13. If you're having troubles, refer to the game.js file you downloaded from manning.com.

Listing 6.10 game.js – HTML5 Game Coding Template

```
if (canvas.getContext){
    var context = canvas.getContext('2d');
    init();
}
```

```
function init() {
    brickInit();
    ballInit();
    padInit();

    draw();
}

function draw() {                                                    #A
    context.clearRect(0,0,canvasW,canvasH);
    background();

    brickDraw();
    ballDraw();
    padDraw();
}
```

#A Draw functions overlay from bottom up

So far in this game development journey, you've created all the core pieces of Breakout. Because nothing moves, the game is as useless as a rod without a reel. In the next section, we'll teach you how to bring your game's static design to life!

6.4 Breathing life into Breakout

Currently your game looks really cool, but it doesn't do anything. Pulling out several different tools from our JavaScript and Canvas toolbox we'll show you how to get your new objects running. In fact, we'll even show you how to integrate physics, and move the paddle with a controller.

6.4.1 Animating game elements

Now that you've created all the needed objects and divided code into init and draw functions, it's time to begin animating. Your moving objects will be visible, but you won't be able to control them via keyboard and mouse until you work through the collision and controller setup in the following sections.

Before we get started, you need to animate Canvas objects by adding *return canvasRun = setInterval(draw, 12);* to the end of the init() function. It's imperative you add the following code snippet at the end so it fires after all the initialization functions have setup their default properties.

```
function init() {
    brickInit();
    ballInit();
    padInit();

    return canvasRun = setInterval(draw, 12);
}
```

Now, you're ready to begin. First, let's make the paddle move from left to right. Next, we'll make the ball move diagonally.

ROWING THE PADDLE

To make the paddle move adjust the x axis each time it's drawn. Making x positive will draw the paddle forward while a negative value will pull it back. First, add an undeclared variable at the top for your paddle's speed. Second, give the variable a value of 4 in padInit() to make things run a little faster than you normally would for players. Finally, add an incremental equation at the beginning of the paddle's draw function. Listing 6.11 puts it all together for you.

Listing 6.11 game.js – Paddle animation

```
var padSpeed;

function padInit() {
    padX = 100;
    padY = 210;
    padW = 90;
    padH = 20;
    padR = 9;
    padSpeed = 4;
}

function padDraw() {
    padX += padSpeed;                                                    #1

    context.beginPath();
    context.moveTo(padX,padY);
    context.arcTo(padX+padW, padY, padX+padW, padY+padR, padR);
    context.arcTo(padX+padW, padY+padH,padX+padW-padR,padY+padH, padR);
    context.arcTo(padX, padY+padH, padX, padY+padH-padR, padR);
    context.arcTo(padX, padY, padX+padR, padY, padR);
    context.closePath();
    context.fillStyle = padGrad();
    context.fill();
}
```

Now, refresh your page. Oh no! The paddle swims off into oblivion due to a lack of a movement limiter #1. The only way to keep your paddle from vanishing is to integrate collision detection, which we'll deal with in the next section. First though, the ball needs to be pitched.

THROWING THE BALL

Making the ball move is almost exactly the same as moving the paddle. Only difference is the x axis and y axis are incremented each time it's drawn. Run the code in listing 6.12. You can tweak the ball's trajectory by adding a few extra points to the Y axis. While optional, such an adjustment will make the ball move at a sharper vertical angle.

Listing 6.12 game.js – Ball animation

```
var ballSX;
var ballSY;

function ballInit() {
```

```
        ballX = 120;
        ballY = 120;
        ballR = 10;
        ballSX = 2;
        ballSY = -2;
    }

    function ballDraw() {
        ballX += ballSX;
        ballY += ballSY;

        context.beginPath();
        context.arc(ballX, ballY, ballR, 0, 2 * Math.PI, false);
        context.fillStyle = ballGrad();
        context.fill();
    }
```

After refreshing, the ball and paddle should fly off the screen and disappear. While their disappearance may leave you depressed and lonely, do not fear! You will soon retrieve them through some basic physics integration.

6.4.2 Detecting collisions

In simple 2D games, using physics could be stated as testing for object overlap. These checks need to be performed each time the interval refreshes and draws an updated set of objects. If an object is overlapping, then you need to activate some logic to handle the situation. For instance, if the ball and paddle overlap that should cause the ball to bounce off.

> **Real Game Physics**
>
> Sad to say, but these physics aren't real in a mathematical sense. If you're interested in learning how to make games more lifelike please see Glenn Fiedler's robust article on game physics at http://gafferongames.com/game-physics/.

Start building collisions detection into Breakout by keeping objects contained inside the play area. After taming objects, you'll focus on using the paddle to bounce the ball at the bricks. After the ball is bouncing back the game's bricks need to be removed when touching a ball. Lastly, create rules to determine when the game is shut down. Let's get started.

MAKING INTERACTIVE OBJECTS

To prevent your mouse and paddle from flying off the screen they'll need to be checked against the <canvas> width and height. Use the canvasW and canvasH variables created earlier to keep your objects in check.

Go to the padDraw() function and replace *padX += padSpeed;* with the equation in the snippet that follows to check if the paddle has a positive X coordinate and is within the <canvas> width. If it is, then update the paddle's X coordinate as normal. If performed correctly, you'll notice that the paddle stops at both the left and right edge.

```
if ((padX > 0) && (padX < (canvasW - padW))) {
```

```
    padX += padSpeed;
}
```

Since the paddle is taken care of, stop the ball from dropping out of gameplay by adding listing 6.13 to the top of your ballDraw() function. Create a test for the ball's Y coordinates and another for the X coordinates. If the ball is overlapping make the speed negative by replacing *ballX += ballSX;* and *ballY += ballSY;*. In addition to reversing the ball you'll need to put it back inside the play area, otherwise you run the risk of it getting stuck at higher movement speeds. Use the following code snippet to make the ball repel off the gameplay area's sides.

Listing 6.13 game.js – Ball edge detection

```
if (ballY < 1) {                                    #A
    ballY = 1;
    ballSY = -ballSY;
}
else if (ballY > canvasH) { }                       #B

if (ballX < 1) {                                    #C
    ballX = 1;
    ballSX = - ballSX;
}
else if (ballX > canvasW) {                         #D
    ballX = canvasW - 1;
    ballSX = - ballSX;
}

ballX += ballSX;
ballY += ballSY;
```

#A Top edge
#B Bottom
#C Left
#D Right

With the ball ricocheting, you'll need to use the paddle to deflect it back towards the bricks. Since the ball changes direction on impact and the paddle stays stationary, put your deflection logic inside ballDraw() by *ballx += ballSX* and *ballY += ballSY;*, as seen in the following snippet. When the ball's x and y coordinates overlap the paddle, bounce the ball in the opposite direction by reversing the y axis with the ballSY variable.

```
if (ballX >= padX && ballX <= (padX + padW) && ballY >= padY && ballY <=
(padY + padW)) {
    ballSY = -ballSY;
}

ballX += ballSX;
ballY += ballSY;
```

If you've played Breakout you'll notice that the paddle's deflection appears to be broken. When the ball hits it isn't dynamically bouncing back based upon where it hits the paddle. To make collisions dynamic, modify the equation to take into account the ball's X axis based upon where it impacted the paddle, as follows:

```
if (ballX >= padX && ballX <= (padX + padW) && ballY >= padY && ballY <=
```

```
(padY + padW)) {
    ballSX = 7 * ((ballX - (padX+padW/2))/padW);
    ballSY = -ballSY;
}
```

REMOVING OBJECTS

With the paddle dynamically generating the ball's trajectory the bricks need to start disappearing. Use the brickDraw() function in listing 6.14 to test if the ball is overlapping when a brick is drawn. If so, reverse the ball's y-axis and set the brick's array data to false so it won't be drawn or interact with the ball any longer.

Listing 6.14 game.js – Removing bricks

```
function brickDraw() {
    for (i=0; i<bRow; i++) {
        for (j=0; j<bCol; j++) {
            if (bricks[i][j] !== false) {
                if (ballX >= brickX(j) && ballX <= (brickX(j) + bW) &&
ballY >= brickY(i) && ballY <= (brickY(i) + bH)) {                    #A
                    bricks[i][j] = false;                             #B
                    ballSY = -ballSY;
                }

                brickColor(i);

                context.fillRect(brickX(j),brickY(i),bW,bH);
            }
        }
    }
}
```

#A Collision test
#B If true brick = false

To finish up Breakout's detection there needs to be a point where a “game over” occurs. In order to stop the game add logic in the ballDraw() function for when the ball goes below the play area. The "game over" occurs in listing 6.15 with a clearInterval() that stops all animation. After that you'll notice two mystery functions that take care of the game over screen (more on screens later).

Listing 6.15 game.js – Ending the game

```
function ballDraw() {
    if (ballY < 1) {
        ballY = 1;
        ballSY = -ballSY;
    }
    else if (ballY > canvasH) {
        clearInterval(canvasRun);                                  #A
        canvasRun = setInterval(goDraw, 12);                       #A
        canvas.addEventListener('click', restartGame, false);      #A
    }

    if (ballX < 1) {
        ballX = 1;
```

```
            ballSX = - ballSX;
        }
        else if (ballX > canvasW) {
            ballX = canvasW - 1;
            ballSX = - ballSX;
        }

        ballX += ballSX;
        ballY += ballSY;

        if (ballX >= padX && ballX <= (padX + padW) && ballY >= padY && ballY
<= (padY + padW)) {
            ballSX = 7 * ((ballX - (padX+padW/2))/padW);
            ballSY = -ballSY;

        }

        context.beginPath();
        context.arc(ballX, ballY, ballR, 0, 2 * Math.PI, false);
        context.fillStyle = ballGrad();
        context.fill();
    }
```

#A Add these lines

Now that you can deflect the ball back towards bricks, players have the ability to defend themselves from the dreaded game over. Well not exactly... we still haven't explained how to let players control the paddle. Whipping up a little bit of jQuery magic we'll show you a few simple code recipes for setting up keyboard and mouse functionality.

6.4.3 Creating keyboard and mouse controls

To create an interactive game experience keyboard or mouse input is required. The sad news is browser support for detecting keyboard events is nothing short of terrible. DOM level 3 is supposed to help standardized keyboard monitoring, but support is currently dodgy.

Although you've gotten away without using it so far, you'll need the jQuery library to detect keyboard events. Download the library from jquery.com and insert it right above the Canvas game's JavaScript file. We highly recommend that you download the minified production file to minimize the game's size (the smaller the size, the faster it loads).

```
<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="game.js"></script>
```

With jQuery locked-in and loaded you can set up a flexible control scheme where the player has options. For the first option you'll create keyboard controls with the left and right arrow keys. For those who wield the power of a mouse, you'll create a JavaScript mouse listener that monitors cursors movement and places the paddle there. Lastly, we'll give you a few tips on best practices for integrating controls based upon techniques that produce happy customers.

MASTERING THE KEYBOARD

To create a keyboard detector you need two new variables near the top of the JavaScript file. One for the left and another for the right keyboard arrows. Think of these monitoring devices as switches for activating left or right movement.

To turn keyboard variables on and off you'll need two jQuery listeners. One that detects when keys pressed down and another that monitors key release. For detecting the keys use keycodes for the most consistent and simple way to detect a user's keyboard input. Based upon the current state of the monitored keys, the paddle is shifted left or right. Dump the code from listing 6.16 at the very bottom of your JavaScript file.

More keycodes!

If you would like to know more about the state of keyboard detection and get a complete list of key codes please see Jan Wolter's article JavaScript Madness: Keyboard Events at http://unixpapa.com/js/key.html.

Listing 6.16 game.js – Keyboard listeners

```
var keyL;                                                    #1
var keyR;                                                    #1

$(document).keydown(function(evt) {
        if (evt.keyCode === 39) {                            #A
                keyL = true;
        }
        else if (evt.keyCode === 37) {                       #B
                keyR = true;
        }
});

$(document).keyup(function(evt) {                            #C
        if (evt.keyCode === 39) {
                keyL = false;
        }
        else if (evt.keyCode === 37) {
                keyR = false;
        }
});
```

#A Check if left arrow key
#B Check if right arrow key
#C Reset keyL and keyR

Refreshing the page you'll notice that the paddle refuses to acknowledge your commands. With the keyL and keyR variables #1 storing keyboard input, the padDraw() function needs to be updated to detect whether to move left or right. The movement function will need to be rewritten into two parts. One that moves the paddle to the right and stops against the right wall and another that does the same for the left. Replace *if ((padX > 0) && (padX < (canvasW - padW))) {}* with the following code snippet to make the paddle react to the variables.

```
if (keyL && (padX < (canvasW - padW))) {
    padX += padSpeed;
}

else if (keyR && padX > 0) {
```

```
        padX += -padSpeed;
    }
```

MANIPULATING THE MOUSE

jQuery does more than hijack keyboard keys; it also makes mouse movement easy to monitor. To get the current mouse location, create a listener at the bottom of your JavaScript file. Note that placing listing 6.17 at the bottom of your JavaScript file will override the keyboard listener you integrated earlier.

Listing 6.17 game.js – Mouse controls

```
var mouseX;
$('#canvas').mousemove(function(e){
    var canvasPosLeft = Math.round($("#canvas").position().left);      #A
    var canvasPos = e.pageX - canvasPosLeft;
    var padM = padW / 2;                                               #B

    if (canvasPos > padM && canvasPos < canvasW - padM) {
        mouseX = canvasPos;
        mouseX -= padW / 2;
        padX = mouseX;
    }
});
```

#A Game position from left in pixels
#B Center paddle on mouse

CONTROL INPUT CONSIDERATIONS

In the past couple years JavaScript keyboard support for applications and websites has grown in leaps and bounds. YouTube, GMail, and other popular applications use keyboard shortcuts to increase user productivity. While allowing users to speed up interaction is useful, it can quickly create problems.

You need to be careful when declaring keyboard keys in JavaScript. You could override default browser shortcuts, remove operating system functionality (copy, paste, etc.), and even crash the browser. The best way to avoid angering players is to stick to the arrow and letter keys. Specialty keys such as the space bar can be used, but overriding shift or caps lock could have unforeseen repercussions. If you must use a keyboard combination or specialty key, ask yourself "Will this cause problems for my application's users in some way?"

When playing videogames people don't want to spend their first 10 minutes randomly smashing keys and clicking everywhere. Put your game's controls in an easy to find location and use concise wording. For instance, placing the controls directly under a game is a great way to help users.

To add a control description to Breakout, add a simple <p> tag directly below <canvas>. It should say "LEFT and RIGHT arrow keys or MOUSE to move" If you really want, you could create a graphical illustration that's easier to see, but for now we'll just use text for simplicity.

```
<canvas id="canvas" width="408" height="250">
    Download FireFox to play this game now!
```

```
</canvas>

<p>LEFT and RIGHT arrow keys or MOUSE to move</p>

<script type="text/javascript" src="jquery.js"></script>
<script type="text/javascript" src="game.js"></script>
```

Congratulations! You've just completed an HTML5 game from beginning to end. You could leave this chapter and be done with it, but why stop here? The best games always have extra features that add more complexity and intrigue as you play. So why not add your own extra material to keep players playing? That's what we'll do next.

6.5 Polishing your game

Although your game is technically complete, it lacks the polish necessary to attract players. Addictive elements such as scoreboards, increased difficulty levels, and an enjoyable user experience are essential. They help to increase game revenue, maximize the number of users, and most importantly, keep people playing. On the other hand, not polishing a game can lead to failure due to broken or badly implemented game mechanics. We personally hate games with badly implemented cameras, unbalanced level difficulty, and clunky user interfaces—and we bet you do, too.

We're going to skyrocket the usefulness of your Breakout game by showing how to polish it to perfection. Start by adding a point system and optional Facebook scoreboard for users. After that you'll create a dynamic leveling system with minor modifications, so users play harder and faster as their skills improve. Then, you'll place the cherry on top of Breakout with opening and closing screens. Lastly, we'll cover the current Canvas gaming engines so you can write your next game faster.

6.5.1 Keeping score

When we were about ten years old (okay, maybe some of us were older!), we used to play Breakout all the time. One of us played on the now-ancient Atari gaming system, another played at Pizza Hut every Friday. We would play over and over to keep raising our high scores. Back then you could only compete with a local community; now, with social media, it's quite easy to put your game's scoreboard online so people can compete on a global scale. But, before you can let users post their high scores online, you'll need to create a system that monitors and records scores.

CREATING A SCORE COUNTER

The simplest way to create a score counter is through a global variable, which stays alive outside of functions. In case the game is reset, the counter value should be set to 0 in the game's init function. After you've declared the default value, wrap a value declaration inside a function and place it inside the init() so everything is reset when the game boots up. You'll find value resets at game startup useful for changing health, magic, and other properties to their default value. To integrate the score counter use listing 6.18 to modify your code.

Listing 6.18 game.js – Score counter variables

```
var score;

function init() {
    hudInit();
    brickInit();
    ballInit();
    padInit();

    return canvasRun = setInterval(draw, 12);
}

function hudInit() {
    score = 0;
}
```

Each time a brick is destroyed, increment the score variable by one. Place the following code snippet in the brickDraw() function to check if the ball has made impact.

```
if (ballX >= brickX(j) && ballX <= (brickX(j) + bW) && ballY >= brickY(i)
&& ballY <= (brickY(i) + bH)) {
    score += 1;                                                    #A
    bricks[i][j] = false;
    ballSY = -ballSY;
}
```

#A Only add this

Since the score exists inside the browser, you'll need to write the current value onto the play area. To show players their scores, create and place a new function inside draw() as displayed in listing 6.19. Call the function hudDraw() and use it exclusively for drawing supplemental game information.

Listing 6.19 game.js – Drawing the updated score

```
function draw() {
    context.clearRect(0,0,canvasW,canvasH);
    background();

    brickDraw();
    ballDraw();
    padDraw();
    hudDraw();
}

function hudDraw() {
    context.font = '12px helvetica, arial';
    context.fillStyle = 'white';
    context.textAlign = 'left';
    context.fillText('Score: ' + score, 5, canvasH - 5);
}
```

STORING HIGH SCORES

With a live updating score counter, users should be given a chance to save their progress when the ball goes out of bounds. Playtomic.com (figure 6.14) allows you to create leader boards, Facebook logins, and fire an event to record high scores. If interested in using score

boards, head on over, sign up for an account, and download their JavaScirpt library. For more information follow the documentation provided on integrating HTML5 leaderboards with your game, as it is a simple but long process.

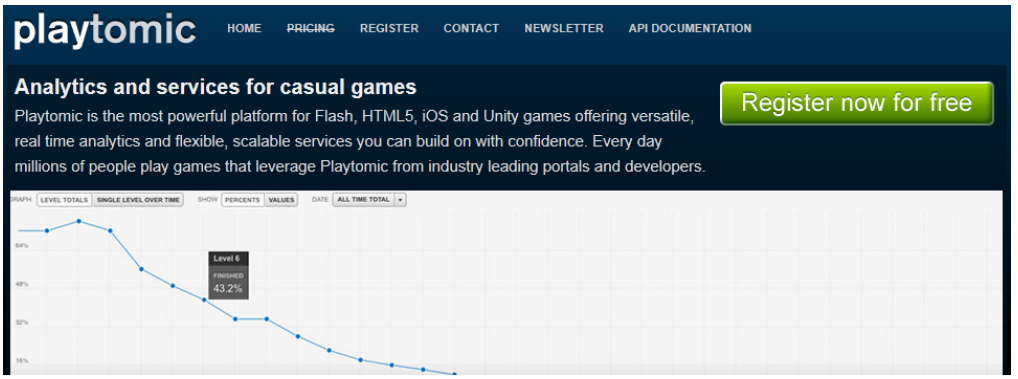

Figure 6.14 Set up leader boards and get advanced analytics for your HTML5 games at playtomic.com. Their API offers a wide variety of features such as event and traffic monitoring. Great for monitoring conversions if your game offers any kind of purchasable goods.

PEOPLE ALWAYS CHEAT

HTML5 is known for many things, but security isn't one of them. Because your game is running in JavaScript, it's quite easy for hackers to manipulate high scores, lives, and other information. Many consider JavaScript's security limitations a huge problem for scoreboards and making income with in-game content.

People will cheat if they can no matter what. On the other hand, if you absolutely have to have some security, two options are available. The simplest is to pass data in increments to your server and have it run checks on the data before storing it. The downside here is it usually requires users to have an account to cross reference data with. The other option is to hide a security code in your JavaScript files that AJAX uses as a handshake with the database to see if the current game is valid. While these two methods will work, they will only temporarily prevent users from hacking your game over time. If you're thinking "I'll have to develop my game in Flash or Java because of security issues," then please realize these systems also have security flaws. There is no 100% secure solution to prevent people from hacking your game. The best solution is to not worry about it if you can.

More than likely you've also heard people say that HTML5's Canvas API is a Flash killer. We're here to tell you that isn't necessarily the case. Because Flash is proprietary in nature, it provides robust security features that Canvas isn't capable of providing yet. Then again Canvas shouldn't be expected to have a robust security suite since it's an open source language. On the other hand Canvas currently appears to have better mobile and Mac OS support. The reason for this is not because one language is necessarily more gifted than the

other, but Adobe doesn't have access to Mac's graphic acceleration API. That and Flash has never been well supported on phones in general. In short, Canvas is not a replacement for Flash and it won't be magically disappearing anytime soon.

6.5.2 Enhanced difficulty

As your game exists right now, there isn't any reward for destroying the bricks. In order to give players an incentive to keep playing, increment the level's difficulty. Perform level ups by incrementing the brick and ball speed variables at each new level. The first step is to create a level variable that starts out at one. The text for this will need to be drawn and the value initialized through the heads up display (HUD) function, as shown in the following listing.

Listing 6.20 game.js – Setting up a leveling system

```
var level;

function hudInit() {
    level = 1;
    score = 0;
}

function hudDraw() {
    context.font = '12px helvetica, arial';
    context.fillStyle = 'white';
    context.textAlign = 'left';
    context.fillText('Score: ' + score, 5, canvasH - 5);
    context.textAlign = 'right';
    context.fillText('Lv: ' + level, canvasW - 5, canvasH - 5);
}
```

Place the level multiplier next to the brick column and ball speed values. After modifying ballSX and ballSY in ballInit() and bRow in brickInit() your code should look like the following snippet.

```
ballSX = .9 + (.4 * level);
ballSY = - .9 - (.4 * level);
bRow = 2 + level;
```

When the level up occurs everything needs to be updated except for the HUD elements. Because of hudDraw() you can't fire the init() function again because it will overwrite the variables for the level and score. Instead, create a new function called levelUp() that fires all of the properties needed.

```
function levelUp() {
    level += 1;
    brickInit();
    ballInit();
    padInit();
}
```

Level completion is detected by counting and checking bricks in an array. Loop through the array during the brickDraw() function to see if they all spit out a false statement. Problem is loops suck up a lot of resources, especially with intervals. Instead, increment a counter each

time a brick is destroyed. At the end of the brickDraw() function run a check to see if the counter equals the total number of bricks. If so, fire the levelUp() function as seen in listing 6.21.

Listing 6.21 game.js – Firing level up

```
var bCount;

function brickInit() {
    bGap = 2;
    bRow = 2 + level;
    bCol = 5;
    bW = 80;
    bH = 15;
    bCount = 0;

    bricks = new Array(bRow);
    for (i=0; i<bRow; i++) {
        bricks[i] = new Array(bCol);
    }
}

function brickDraw() {
    for (i=0; i<bRow; i++) {
        for (j=0; j<bCol; j++) {
            if (bricks[i][j] !== false) {
                if (ballX >= brickX(j) && ballX <= (brickX(j) + bW) &&
ballY >= brickY(i) && ballY <= (brickY(i) + bH)) {
                    score += 1;
                    bCount += 1;
                    bricks[i][j] = false;
                    ballSY = -ballSY;
                }

                brickColor(i);

                context.fillRect(brickX(j),brickY(i),bW,bH);
            }
        }
    }

    if (bCount === (bRow * bCol)) {
        levelUp();                                                    #A
    }
}
```

#A Level up occurs here

With a great leveling system users now have more reason to play. More incentive to play equals happier players, longer gaming time, and more likelihood that your content will be shared. Sadly, people might not be so happy since the number of brick rows can infinitely increase (hindering player's fun). In order to stop bricks from taking over the screen, create a function that limits the level number using the code in listing 6.22.

Listing 6.22 game.js – Preventing level up brick overflow

```
function levelLim(lv) {
    if (lv > 5) {
        return 5;
    }
    else {
        return lv;
    }
}
```

To implement your new level limiting function replace *bRow = 2 + level;* with *bRow = 2 + levelLim(level);*. After that you shouldn't have any more brick overflow problems.

6.5.3 Opening and closing screen

When a user loads up your game as it exists now, they're forced to start playing immediately or lose. Forcing players to play is intrusive and imposing since it doesn't let them initiate. In order to let the user start up the game, create a welcome screen that starts on click via jQuery listener.

Figure 6.15 A simple welcome screen that initiates the game on click through a jQuery listener. All of the text and black backdrop is created through Canvas.

The first step to making a simple initiation screen like the one you see in figure 6.15 is to create a function that overlaps everything during the draw phase. It needs to have a background with the width and height of the canvas to cover everything up. On top of the background it should say "BREAKOUT" and "Click To Start." Use listing 6.23 to draw your welcome screen.

Listing 6.23 game.js – Creating the welcome screen and listener

```
function welcomeDraw() {
    context.fillStyle = 'black';
    context.fillRect(0, 0, canvasW, canvasH);

    context.fillStyle = 'white';
    context.textAlign = 'center';
    context.font = '40px helvetica, arial';
    context.fillText('BREAKOUT', canvasW/2, canvasH/2);

    context.fillStyle = '#999999';
    context.font = '20px helvetica, arial';
    context.fillText('Click To Start', canvasW/2, canvasH/2 + 30);
}
```

With the welcome screen stored in a function, you'll need to call it with a click event listener. This is the same kind of listener that you setup with jQuery previously to monitor mouse movements. In addition to setting up the listener, you'll need to remove it after the game starts to prevent users from accidentally resetting the game with a mouse click.

Listing 6.24 game.js – Creating the welcome screen and listener

```
if (canvas.getContext){
    var context = canvas.getContext('2d');

    welcomeDraw();
    canvas.addEventListener('click', runGame, false);
}

function runGame() {
    canvas.removeEventListener('click', runGame, false);
    init();
}
```

Figure 6.16 Game over screen with a second chance at life. It helps to create an easy "Try again?" format that lets users keep on playing without requiring a complete page refresh.

With a welcome screen in place the user can seamlessly play at will until the ball disappears. Thinking a few sections back you might remember that we had you setup a few lines of mystery code for when the ball goes out of bounds. Below you'll find one of the referenced functions in listing 6.25 to create the game over screen.

Listing 6.25 game.js – Game over screen

```
function goDraw() {
        context.fillStyle = 'black';
        context.fillRect(0,0,canvasW,canvasH);

        context.fillStyle = 'red';
        context.textAlign = 'center';
        context.font = '40px helvetica, arial';
        context.fillText('Game Over', canvasW/2, canvasH/2);

        context.fillStyle = '#999999';
        context.font = '20px helvetica, arial';
        context.fillText('Click To Retry', canvasW/2, canvasH/2 + 30);
}
```

The other line of code that fires when the ball drops off the edge is a listener. On click that listener fires to the function in listing 6.26 to detect if the game should be restarted. If so the game's interval is cleared and the init() function is run to restart the game.

Listing 6.26 game.js – Game over listener

```
function restartGame() {
        canvas.removeEventListener('click', restartGame, false);
        clearInterval(canvasRun);
        init();
}
```

6.5.4 Code libraries you should consider

By completing Breakout, you're now also capable of coding your own games in Canvas. On the other hand, it probably took a while to code everything out from scratch. To help save time and money on projects, we highly recommend that you use a JavaScript library. For example, with one of our library suggestions you could write Breakout in 100 lines or less. On the other hand, if you had done that, you wouldn't understand the core concepts of how all the game elements work together. Currently three major libraries exist on the market for Canvas, each with a specific audience in mind.

Figure 6.17 Code libraries like ImpactJS allow you to create complex games in significantly less time than coding a game from scratch.

AKIHABARA

Named after a major Tokyo shopping area for anime and electronics, Akihabara has earned a place above other HTML5 libraries. It's targeted at creating 8 bit game clones, but has also

done some more impressive games like a Guitar Hero spinoff for the keyboard. The library itself is free to use and sits under a GPL2/MIT license.

ROCKET ENGINE

Developed by Rocket Pack, the Rocket Engine is aiming to deliver a canvas code library similar to the Unreal Tournament Engine. If you aren't familiar with Unreal, a large number of popular videogames have been developed with it, such as Mass Effect, Gears of War, and Bioshock. This powerful idea is so intriguing that Disney purchased the company. The library promises to deliver flexible tools for massive multiplayer, Facebook, and even single player games. Currently the engine isn't open to the public, but their first game built with the engine called Warimals is available to play.

IMPACTJS

Known as the Impact JavaScript Engine (figure 6.17), this HTML5 library is one of the fastest and most effective libraries out there. Shinobi Content, one of our firms, uses it exclusively for all Canvas based games. It has documentation that's slowly rising up near jQuery's and video tutorials to get you up and running. The best feature though is the system's built in level editor. The only catch is it costs $99 per license. While that may seem steep for some, consider that the core developer will personally answer your questions on the forums. He has answered several of our questions and even added new features within days of a request.

Convert HTML5 games to mobile apps

Want to turn your new HTML5 game into a mobile application for Android, IOS, and other systems? Check out appMobi.com and PhoneGap.com for powerful conversion tools that give you access to all major mobile devices.

6.6 Summary

Canvas itself is not limited to a small box for video games; you can make it as large as you want for applications and use it inside websites. Thinking out of the box you can create interactive backgrounds, headers, banners, and more. For instance, you could make an interactive experience with a footer in which users play a game of breakout destroying footer elements once they've initiated the game.

While you did play with the Canvas animation element, we've barely touched on what it's capable of. You could even animate a small film with it, which will become more possible as GUI tools to interact with Canvas become accessible. In the meantime you can make pages react to mouse position location or activate animation sequences based upon mouse clicks or hover. Remember, you have all of the JavaScript functions at your fingertips and jQuery makes event listeners easy to access.

While 2D games can be fun to make, they aren't exactly making record sales in the game market. In addition, most 2D based game and animation studios have closed down or converted over to 3D. If browser interfaces want to compete with professional tools such like 3D Studio Max, Maya, and After Effects, then 3D must be capable in the browser. On the

other hand, 2D applications can be fairly cheap to produce and casual gaming on the internet has grown tremendously in the past ten years, which is resulting in high demand for 2D games (especially on mobile devices). While you can create all of your 2D games and applications in Canvas, you may also want to consider using SVG. It has an incredibly large set of features and puts Canvas to shame when it comes to creating graphics—and we're going to explore it in more detail next.